Extracting Parallelism from Legacy Sequential Code Using Transactional Memory

Mohamed M. Saad

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
Anil Kumar S. Vullikanti
Paul E. Plassmann
Robert P. Broadwater
Roberto Palmieri
Sedki Mohamed Riad

May 25, 2016
Blacksburg, Virginia

Keywords: Transaction Memory, Automatic Parallelization, Low-Level Virtual Machine, Optimistic Concurrency, Speculative Execution, Legacy Systems, Age Commitment Order, Low-Level TM Semantics, TM Friendly Semantics

Extracting Parallelism from Legacy Sequential Code Using Transactional Memory

Mohamed M. Saad

(ABSTRACT)

Increasing the number of processors has become the mainstream for the modern chip design approaches. However, most applications are designed or written for single core processors; so they do not benefit from the numerous underlying computation resources. Moreover, there exists a large base of legacy software which requires an immense effort and cost of rewriting and re-engineering to be made parallel.

In the past decades, there has been a growing interest in automatic parallelization. This is to relieve programmers from the painful and error-prone manual parallelization process, and to cope with new architecture trend of multi-core and many-core CPUs. Automatic parallelization techniques vary in properties such as: the level of paraellism (e.g., instructions, loops, traces, tasks); the need for custom hardware support; using optimistic execution or relying on conservative decisions; online, offline or both; and the level of source code exposure.

Transactional Memory (TM) has emerged as a powerful concurrency control abstraction. TM simplifies parallel programming to the level of coarse-grained locking while achieving fine-grained locking performance. This dissertation exploits TM as an optimistic execution approach for transforming a sequential application into parallel. The design and the implementation of two frameworks that support automatic parallelization: Lerna and HydraVM, are proposed, along with a number of algorithmic optimizations to make the parallelization effective.

HydraVM is a virtual machine that automatically extracts parallelism from legacy sequential code (at the bytecode level) through a set of techniques including code profiling, data dependency analysis, and execution analysis. HydraVM is built by extending the Jikes RVM and modifying its baseline compiler. Correctness of the program is preserved through exploiting Software Transactional Memory (STM) to manage concurrent and out-of-order memory accesses. Our experiments show that HydraVM achieves speedup between 2×-5× on a set of benchmark applications.

Lerna is a compiler framework that automatically and transparently detects and extracts parallelism from sequential code through a set of techniques including code profiling, instrumentation, and adaptive execution. Lerna is cross-platform and independent of the programming language. The parallel execution exploits memory transactions to manage concurrent and out-of-order memory accesses. This scheme makes Lerna very effective for sequential applications with data sharing. This thesis introduces the general conditions for embedding any transactional memory algorithm into Lerna. In addition, the ordered version of four state-of-art algorithms have been integrated and evaluated using multiple benchmarks including RSTM micro benchmarks, STAMP and PARSEC. Lerna showed great results with average 2.7× (and up to 18×) speedup over the original (sequential) code.

While prior research shows that transactions must commit in order to preserve program semantics, placing the ordering enforces scalability constraints at large number of cores. In this dissertation, we eliminates the need for commit transactions sequentially without affecting program consistency. This is achieved by building a cooperation mechanism in which transactions can forward some changes safely. This approach eliminates some of the false conflicts and increases the concurrency level of the parallel application. This thesis proposes a set of commit order algorithms that follow the aforementioned approach. Interestingly, using the proposed commit-order algorithms the peak gain over the sequential non-instrumented execution in RSTM micro benchmarks is 10× and 16.5× in STAMP.

Another main contribution is to enhance the concurrency and the performance of TM in general, and its usage for parallelization in particular, by extending TM primitives. The extended TM primitives extracts the embedded low level application semantics without affecting TM abstraction. Furthermore, as the proposed extensions capture common code patterns, it is possible to be handled automatically through the compilation process. In this work, that was done through modifying the GCC compiler to support our TM extensions. Results showed speedups of up to 4× on different applications including micro benchmarks and STAMP.

Our final contribution is supporting the commit-order through Hardware Transactional Memory (HTM). HTM contention manager cannot be modified because it is implemented inside the hardware. Given such constraint, we exploit HTM to reduce the transactional execution overhead by proposing two novel commit order algorithms, and a hybrid reduced hardware algorithm. The use of HTM improves the performance by up to 20% speedup.

This work is supported by VT-MENA program.

Dedication

To my wife, my parents, and my grandma.

Acknowledgments

I would like to thank my advisor, Dr. Binoy Ravindran, for all his help and support. Also, I would like to thank my friend, mentor and co-advisor, Dr. Roberto Palmieri, for his guidance, dedication and efforts. I would also like to thank my committee members: Dr. Robert P. Broadwater, Dr. Anil Kumar S. Vullikanti, Dr. Paul E. Plassmann and Dr. Sedki Mohamed Riad, for their guidance, feedbacks and advice. It is a great honor for me to have them serving in my committee.

Many thanks to all my friends in the Systems Software Research Group for their support, assistance and cooperation. They include, Dr. Ahmed Hassan, Dr. Mohamed Mohamedin, Dr. Sandeep Hans, Dr. Sebastiano Peluso, Dr. Alexandru Turcu, Joshua Bockenek, Marina Sadini, and many others. Also, special thanks to Dr. Mohamed E. Khalefa for the fruitful discussions and his valuable inputs that had considerable influence in shaping this work.

Finally, I would like to thank my wife, Shaimaa, for her endless love, understanding and patience. Also, I would like to thank my parents for their encouragement and support.

Contents

List of Figures

List of Tables

Acronyms

ACO:	Age-based Commit Order
Advisor XE:	Intel Parallel Advisor
AOS:	Adaptive Optimization System
BiSort:	Bitonic Sort
BTH:	Burning Tickets Algorithm
CFG:	Control Flow Graph
CM:	Contention Management
CMP:	Chip Multiprocessor
DASTM:	Dependency Aware STM model
DBMS:	Database Management Systems
ETL:	Encounter-Time Locking
GC:	Garbage Collector
GCC:	GNU Compiler Collection
HJM:	Heath-Jarrow-Morton
HTM:	Hardware Transactional Memory
HyTM:	Hybrid Transactional Memory
ILP:	Instruction-Level Parallelization
IR:	Intermediate Representation
Jikes RVM:	Jikes Research Virtual Machine
JNI:	Java Native Interface
JVM:	Java Virtual Machine
LLVM:	Low-Level Virtual Machine
LRU:	Least Recently Used
LSA:	Lazy Snapshot Algorithm
MC:	Monte Calro simulation
MMTK:	The Memory Manager Toolkit
MPI:	Message Passing Interface
MSSP:	Master/Slave Speculative Parallelization
MST:	Minimum Spanning Tree
NOrec:	No Ownership Records Algorithm
NUMA:	Non-Uniform Memory Access
OLTP:	On-Line Transaction Processing
OpenMP:	Open Multi-Processing
OUL:	Ordered Undo Logging Algorithm
OUL-Steal:	OUL Lock-Steal Algorithm
OWB:	Ordered Write Buffer Algorithm
OWB-RH:	OWB Reduced Hardware Algorithm
PLPP:	Pattern Language for Parallel Programming
RAW:	Read-After-Write
RTM:	Restricted Transactional Memory

SMTX:	Software Multi-threaded Transactions
S-NOrec:	Semantic NOrec Algorithm
SSA:	Static Single Assignment form
STAMP:	Stanford Transactional Applications for Multi-Processing
S-TL2:	Semantic Transactional Locking 2 Algorithm
STM:	Software Transactional Memory
TCM:	Transaction Commit Manager
TFH:	Timeline Flags Algorithm
TL2:	Transactional Locking 2 Algorithm
TLS:	Thread-Level Speculation
TM:	Transactional Memory
TMS1:	Transactional Memory Specification 1
TPC-C:	Transaction Processing Performance Council
TSP:	Traveling Salesman Problem
WAR:	Write-After-Read
WAW:	Write-After-Write

Chapter 1

Introduction

In the last decade, parallelism has gained a lot of attention due to the physical constraints which prevent the increase of processor operating frequency. The runtime of a program is measured by the time required to execute its instructions. Decreasing the runtime requires reducing the execution time of a single instruction, which implies increasing the operating frequency. From the mid 1980s until the mid 2000s[1] this approach, namely *frequency scaling*, was the dominant force to improve programs runtime in commodity processor performance. However, operating frequency is proportional to the power consumptions and consequently heat generation. These obstacles put an end to the era of frequency scaling and force the chip designers to find an alternative approach for improving the performance of applications.

Gordon E. Moore made an empirical observation that the number of transistors in a dense integrated circuit has doubled approximately every two years. With the power consumption issues, these additional transistors are moved to add extra hardware that have made the use of multicore processors the norm for microarchitecture chip design. Wulf *et. al.* [187] report that the rate of improvement in microprocessor speed exceeds the rate of improvement in memory speed, namely *memory wall*. This wall constraints the performance of any program running on a single processing unit.

These combined factors (i.e., power consumption, the availability of extra hardware resources, and memory wall) made *parallelism*, which primarily has been employed for long time in high-performance computing, appear as an appealing alternative.

Parallelism is the execution of a sequential application simultaneously on multiple computation resources. The application is divided into multiple sub-tasks that can run in parallel. The communication between sub-tasks defines the parallelism granularity. An application is *embarrassingly parallel* when the communications between its sub-tasks are rare while an application with a lot of sub-task communications exhibits fine-grained parallelism. The

[1] At year 2004, because of the increase in processor power consumption Intel announced the cancellation of its Tejas and Jayhawk processors, the successors processors families for Pentium 4 and Xeon respectively.

maximum possible speedup of a single program as a result of parallelization is known as Amdahl's law. This law defines the relation between speedup and the time needed for the sequential fraction of the program. On the other hand, the vast majority of the applications and algorithms are designed or written for single core processors (often intentionally designed to be sequential to reduce development costs, while exploiting Moore's law of single-core chips).

1.1 Motivation

Many organizations with enterprise-class legacy software are increasingly faced with a hardware technology refresh challenge due to the ubiquity of the Chip Multiprocessor (CMP) hardware. This problem is apparent when legacy codebases run into several million LOC and are not concurrent. Manual exposition of concurrency is largely non-scalable for such codebases due to the significant difficulty in exposing concurrency and ensuring the correctness of the converted code. In some instances, sources are not available due to proprietary reasons, intellectual property issues (of integrated third-party software), and organizational boundaries. Additionally, adapting these programs to exploit hardware parallelism requires a (possibly) massive amount of software rewriting operated by skilled programmers with knowledge and experience of parallel programming. They must take care of both high-level aspects, such as data sharing, race conditions, and parallel sections detection, and low-level hardware features, such as thread communication, locality, caching, and scheduling. This motivates techniques and tools for *automated concurrency refactoring*, or *automatic parallelization*, and designing new programming languages to aid writing parallel programs, or *manual parallelization*.

1.1.1 Manual Parallelization

Designing a parallel program requires both identifying and implementing parallelism. This gained a significant research interest for helping programmers to identify and to develop parallel programs. Identifying parallelism requires a deep understanding of the sequential program to convert and the problem domain. In [118], Pattern Language for Parallel Programming (PLPP) was proposed to formalize high-quality solutions to frequently occurring problems in the parallelization domain. PLPP helps programmers to find concurrency; to build a parallel structure such as divide and conquer, pipeline, decomposition, and event based coordination; and to support these structures using techniques like master/worker, fork/join, shared queues, distributed arrays and loop parallelism. Intel Parallel Advisor [1] (or Advisor XE) automates the manual parallelization by proving a shared-memory threading design and prototyping tool. It helps programmers detect parallel sections and compare performance using different threading models. Additionally, it finds data dependencies and enables eliminating them, if possible.

On the other hand, different languages have been designed to facilitate parallel programming development. For example, OpenMP [51] is a compiler extension for the C, C++ and Fortran languages that supports adding parallelism to existing code, mainly loops, without significant changes. Cilk [27] is C-based runtime for multithreaded parallel programming using the fork/join model. Cilk represents the program as a directed graph of non blocking code snippets. Each node in the graph executes its code within a separate thread. A node may spawn children nodes, and its code runs concurrently with its children's code (non blocking execution). However, in order to receive children's return values, the parent node needs to spawn a *successor* node. A different approach was introduced by the Atomos [40] language where parallel sections are organized as *transactions*. Atomos programming model supports strong atomicity by providing `watch` and `retry` constructs to handle conflicting parallel sections.

Despite all the aforementioned techniques, tools and programming languages, the procedure of manual parallelization is still costly, error pruning, time consuming, and inapplicable in some situation where the source code is not available.

1.1.2 Automatic Parallelization

Automatic parallelization simplifies the life of programmers, especially those not extensively exposed to the nightmare of developing efficient concurrent applications. It also allows a growing number of (even legacy) systems to benefit from the presently available cheap hardware parallelism. Automatic parallelization targets programming transparency; the application is coded as sequential and the specific methodology used for handling concurrency is hidden to the programmer.

Automatic parallelization has been extensively studied during the past decades [68, 79, 146, 181, 176, 43, 45, 65, 142, 111, 186, 120]. Past efforts on automatic parallelization can be broadly classified into speculative (optimistic execution) and non-speculative (pessimistic execution) techniques.

Non-speculative techniques usually come in the form of compiler support where static analysis of the code is employed. For example, *alias analysis* [47] detects if multiple references point to the same memory location. *Memory dependence analysis* [17, 184] analyzes a memory operation and extracts the preceding memory operations that depend on it. *Points-to analysis* [175] identifies, with some assumptions on the address type, which memory locations are referred to by pointers. The *polyhedral analysis* [29, 74] uses an abstract mathematical representation to analyze the memory access patterns. Unlike the dependency analysis where each node in the dependency graph represents one *statement* in the source program, in the polyhedral model each point corresponds to one *statement instance*. This permits building a distance vector between instances which captures the carried dependencies. That way several loop transformations can be applied to enhance the parallelism such as: tiling, index set splitting, loop fusion and nested loop regeneration.

Using these analyses, and others, helps non-speculative techniques in identifying sections of the code with no memory dependencies. These *independent* sections of the code are eligible to run in parallel while preserving a consistent view of memory. However, these techniques do not translate well with low-level memory operations (e.g., pointer arithmetic, unions and functions pointer) in C and C++ programming languages.

On the other hand, speculative techniques exploits optimistic executions with a compensating actions to recover from invalid operations (e.g., memory conflicts). Among speculative techniques, two appealing primary approaches exist: thread-level speculation (TLS) and transactional memory (TM). TLS runs code speculatively, usually through hardware, and eventually the correct order is determined. The evaluated operations are detected to be correct or not. Incorrect results are discarded and the thread is restarted. TLS uses processor cache as a buffer for thread memory changes. Cache coherence protocols are overloaded with the TLS monitoring algorithm. Squashing a thread is done by aborting the thread and discarding the cache contents while committing thread changes is done by flushing the cache to main memory. Parallelization using thread-level speculation (TLS) has been extensively studied using both hardware [177, 102, 80, 44] and software [146, 111, 146, 43]. It was originally proposed by Rauchwerger *et. al.* [146] for identifying and parallelizing loops with independent data access – primarily arrays. The common characteristics of TLS implementations are: they largely focus on loops as a unit of parallelization; they mostly rely on hardware support or changes to the cache coherence protocols; and the parallel sections are usually small (e.g., innermost loops). An example of TLS automated parallelizing compilers is POSH [111]. POSH focuses on loops and subroutines as unit of parallelization, and employs a profiling phase to discard ineffective sections (non hot-spot portions of the code). Using a simulated TLS CMP with 4 superscalar cores, POSH achieves an average speedup by 1.3 on SPECint2000 applications.

The second speculative approach is Transactional Memory (TM). Transactions were originally proposed by Database Management Systems (DBMS) to guarantee atomicity, consistency, data integrity, and durability of operations manipulating shared data. This synchronization primitive has been recently ported from DBMS to concurrent applications (by relaxing durability), providing a concrete alternative to the manual implementation of synchronization using basic primitives such as locks. This new multi-threading programming model has been named Transactional Memory [93]. TM has emerged as a powerful concurrency control abstraction [83, 107] that permits developers to define critical sections as simple *atomic* blocks, which are internally managed by the TM itself. It allows the programmer to access shared memory objects without providing the mutual exclusion by hand, which inherently overcomes the drawbacks of locks. As a result, with TM the programmer still writes concurrent code using threads, but the code is now organized so that reads and writes to shared memory objects are encapsulated into atomic sections. Each atomic section is a *transaction*, in which the enclosed reads and writes appear to take effect instantaneously. Transactions speculatively execute, while logging changes made to objects– e.g., using an undo-log or a write-buffer. When two transactions conflict (e.g., read/write,

write/write), one of them is aborted and the other is committed, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes–e.g., undoing object changes using the undo-log (eager), or discarding the write buffers (lazy). Besides a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking implementations [59, 33], and allows composability [84], which enables multiples nested atomic blocks to be executed as a single all-or-nothing transaction. Multiprocessor TM has been proposed in hardware (HTM), in software (STM), and in hardware/software combination. TM's adoption is growing in the last years, specially after the integration with the popular *GCC* compiler (from the release 4.7), and due to the integration into the embedded cache-coherence protocol of commodity processors, such as Intel [99] and IBM [35], which naively allows transactions to execute directly on the hardware. Given its easy-to-use abstraction, TM would seem the missing building block enabling the (transparent) parallelization of sequential code by automatically injecting atomic blocks around parallel sections. STMlite [120] presents a lightweight STM implementation customized for automatic loop parallelization. Loop iterations are divided into chunks and distributed over available cores. An outer loop is generated around the original loop body to manage parallel execution between different chunks, and a centralized Transaction Commit Manager (TCM) controls the transactions commit in the program's chronological order. Unfortunately, such high-level abstraction comes with a price in terms of performance cost, which easily leads to a parallel code slower than its sequential version (e.g., in [38, 166] a performance degradation of between 3 to 7 times has been shown).

Most of the methodologies, tools and languages for parallelizing programs target scientific and data-parallel, computation-intensive applications, where actual data sharing is very limited and the dataset is precisely analyzed by the compiler and partitioned so that parallel computation is possible. Examples of those approaches include [134, 153, 111, 94]. Alternatively, automatic parallelization using TLS requires special hardware support, while TLS has not made it into the mainstream multicore systems yet. Finally, despite the flexibility and ease of use of TM, the primary drawback of using it for parallelization is the significant overhead of code execution and validation [38, 166]. Previous work on using TM for parallelization [73, 178] relied primarily on the programmer for specification of parallel sections and definition of ordering semantics. The key weaknesses with programmer reliance are:

1. It requires a full understanding of the software (e.g., the algorithm implementation and input characteristics) and mandates the existence of the source code;
2. It does not take into account TM characteristics and factors of overhead; and
3. It uses TM as a black box utility for preserving data integrity.

In contrast, the solutions proposed in this thesis target common programs and analyze the code in its intermediate representation without the need for the original source code, and without using custom hardware support. This makes us independent of the language in

Property	Manual [118, 1]	Automatic			
		Non-Speculative [74]	Speculative		
			TLS [177, 102, 80, 44]	TM [120, 73]	This Dissertation
No custom hardware (commodity machines)	✓	✓		✓	✓
Tolerate non trivial dependencies	✓		✓	✓	✓
White-box TM model (cooperative model, ordering algorithms)	–	–	–		✓
Automatic parallelization (no user intervention)		✓	✓	✓	✓
Employs static analysis		✓			✓
Language independence / Intermediate representation			✓		✓
Employs application semantics	✓				✓

Table 1.1: Comparison with existing techniques.

which the code was written and does not enforce exposure of the source code to the proposed framework, especially when it is not available. Additionally, the parallel version of the code produced by the proposed framework runs on commodity machines.

This dissertation is fundamentally different from past STM-based parallelization works in that it benefits from static analysis for reducing transactional overheads and automatically identify parallel sections (i.e., traces or loops) by compile- and runtime program analysis techniques, which are then executed as transactions. Additionally, this work targets arbitrary programs (not just recursive ones as [32] does), is entirely software-based (unlike [32, 66, 56, 182]), and does not require program source code. Furthermore, this dissertation proposes a cooperative transactional execution model where atomicity is relaxed with preserving the original sequential semantics. Finally, it extends TM constructs to get use of low-level application semantics to increase the parallelism.

Table 1.1 summarizes some of the main properties of this dissertation and compares them against existing techniques. Unlike other techniques, TLS usually uses hardware to run the code speculatively [177, 102, 80, 44]. This makes TLS independent of the underlying source code or the programming language. This dissertation achieves a similar level of programming language-independence by relying on intermediate representations of the source code (e.g., JVM bytecode, LLVM bytecode, GCC Gimple). Non speculative solutions [74] rely on strong guarantees of data or control dependencies provided by static analysis [47, 17, 175, 29]. These guarantees limit parallelization where non trivial dependencies exist (e.g., pointer

operations, global state accesses). This dissertation employs static analysis for a different purpose – i.e., reducing TM overhead. On the other hand, speculative execution techniques using TM [120, 73] use TM as a black box, however, TM is designed for concurrency control rather than parallelization. The use of TM for parallelization requires some adaption to the classical TM model (i.e., white-box TM model), which is done as a part of this work. Finally, manual parallelization techniques [118, 1] open the door for the programmer to rewrite (refactor) the code while preserving application semantics. This dissertation shares the same sweet-spot of utilizing application semantics for parallelization through a novel semantic-based TM extension.

1.2 Contributions

This thesis tries to bridge the gap between the parallelism of existing multicore architectures and the sequential design of most (including existing) applications. This is done by presenting a set of techniques for extracting parallelism from sequential programs for speculative parallel thread execution on multiprocessor architectures and "thread transactification" for managing concurrent and out-of-order memory accesses. The techniques include code profiling, alias analysis, data dependency analysis, execution analysis at the intermediate representation level, cooperative transactional execution, and exploiting low level semantics of programs. The dissertation makes the following three main contributions.

Automatic Parallelization Frameworks

This dissertation introduces the design and the implementation of two automatic parallelization frameworks: HydraVM and Lerna. Both of HydraVM and Lerna require no programmer intervention, they are completely automated, and do not need to expose source code; unlike previous approaches for parallelization that require programmer to identify parallel sections or additional information to help the parallelization process. The common technique used in the proposed two implementations is the use of Transactional Memory (TM) as an optimistic execution approach for transforming a sequential application into parallel "blindly", meaning without external interventions. Using the proposed frameworks, user benefits from TM seamlessly by running sequential programs and enjoys a safe parallel execution. Under-the-hood, the frameworks handle the problems of detecting parallel code snippets, concurrent memory access, synchronizations and resources utilization.

Unlike Lerna, HydraVM exploits dynamic techniques for profiling, parallel code detection and adaptive execution of the program. Besides, HydraVM targets *trace* as a unit of parallelism, while Lerna focuses on *loop* parallelization. From the experience with HydraVM, it is learnt about parallel patterns and bottlenecks in the programs. First, relying only on dynamic analysis introduces an overhead that can be easily avoided through pre-execution

static analysis phase. Second, adaptive design should not be limited to the architecture (e.g., runtime recompilation), but it could be extended to: selecting best number of worker threads, assignment of parallel code to threads, and the depth of speculative execution. Last, except recursive calls, most of the detected parallel traces at HydraVM was primarily loops. Keeping these goals in sight, Lerna was designed as a compiler framework. With Lerna, the sequential code is transformed into parallel with best-efforts TM support to guarantee safety and to preserve ordering. Lerna produces a native executable, yet adaptive; thanks to Lerna runtime library that orchestrates and monitor the parallel execution. The programmer can interact with Lerna to aid the static analysis for the sake of producing a less overhead program.

HydraVM and Lerna are the first set of compiler/run-time infrastructures that advance the state-of-the-art [120] by: employing static analysis to reduce TM overhead and to allow accessing some local memory variables on stack, providing a vehicle to run different TM algorithms for parallelization (including STMLite [120]), and presenting an adaptive model that maintains the feedbacks collected from the execution to optimize the transactional execution and to recover from misprofiling situations.

Commit-Order TM Algorithms

Parallelization using TM requires concurrency control algorithms that ensure a specific (predefined) commit-order of the transactions injected in the runtime framework. Transaction ordering intuitively means considering not just the set of transactions as input of the problem, but also the specific commit-order that must be enforced for them. Such a formulation inherently brings up a fundamental trade off between the level of parallelism achievable, given the need of committing in-order, and the performance of the single threaded execution without any software instrumentation (which is rather needed to prevent conflicts when running in parallel). TM algorithms have been designed to solve the classical concurrency problems where a set of transactions is invoked in parallel and a history of their operations is built to let them commit.

In this thesis, a set of commit-order TM algorithms is proposed: Ordered Undo Logging Algorithm (OUL), Ordered Write Buffer Algorithm (OWB), OUL Lock-Steal Algorithm (OUL-Steal), and OWB Reduced Hardware Algorithm (OWB-RH), that are designed with any eye on the commit-order as a fundamental system requirement. It is shown that even in the presence of data conflicts, the proposed algorithms are able to outperform single-threaded execution, and other baseline and specialized state-of-the-art competitors, significantly. OUL, OWB, OUL-Steal and OWB-RH are the first set of TM algorithms that exploit cooperation between transactions for the sake of reducing the in order commit waiting time. OWB, and its RH variant, are the first algorithms that support Transactional Memory Specification 1 (TMS1) [62], a weaker consistency condition than opacity [76, 77], the most popular consistency condition for TM. TMS1 has been proved to be sufficient to guarantee safety [13] in

the parallelization model, as is the case with opacity.

Low Level TM Semantics

Analyzing the results produced by executing the aforementioned commit-order algorithms inside the proposed automatic parallelization frameworks, it is clear that there is still a gap in performance that is difficult to fill without having a deeper knowledge of the semantics of the original (sequential) application.

This thesis proposes a solution that increases the level of semantics that can be captured automatically, therefore improving performance further. The proposed extensions also boost the capabilities of the classical transactional memory programming model. In fact, historically, the TM model defines two language/library constructs for reading and writing memory addresses. As a common pattern, each transaction maintains its own read-set and write-set to detect conflicts with other concurrent transactions. Two concurrent transactions are said to be conflicting if they access the same address and at least one access is a write. However, in some situations conflicting transactions can safely commit and still preserve the application semantics, which means that the conflict triggered by the TM framework is a "false conflict" at the semantic level. As a part of this thesis, the classical TM primitives were extended by identifying TM-friendly semantics and proposing an approach to inject them in the current TM algorithms and frameworks. Furthermore, the proposed extensions were integrated in GCC to provide a full compiler support. The proposed TM-friendly semantic primitives are the first such extensions that allow both the compiler-support and capturing application low-level semantics. Furthermore, this extension can be deployed as a TM library without affecting TM generality and with a minimum learning curve for the programmers because all the extensions map to known programming language primitives.

1.2.1 HydraVM

This is a Java virtual machine based on Jikes RVM. The user runs its sequential program (Java classes) using the virtual machine, and internally the bytecode is instrumented at runtime to profile the execution paths. The profiling detects which places of the code are suitable for parallelization and does not have data dependency. Next, the bytecode is reconstructed by extracting portions of the code to run as separate threads. Each thread runs as a transaction, which means it operates on its private copy of memory. Conflicting transactions are aborted and retried while successful transactions commit its changes at the chronological time of the code it executes.

HydraVM [160] inherits the adaptive design of Jikes RVM. The profiling and code reconstruction continue while the program is running. It monitors the performance and abort rate of transformed code and uses this information to repeatedly reconstructs new versions of the code. For example, assume a loop is parallelized and each iteration runs in a separate

transaction. When every three consecutive iterations of loop conflict with each other, then a better reconstruction is to combine them and makes the transaction runs three iterations instead one. It worth noting that the reconstruction process occurs at runtime by reloading the classes definition.

Finally, ByteSTM was developed as a software transactional memory implementation at the bytecode level. ByteSTM allows access to low-level memory (e.g., registers, thread stack), so it is possible create a memory *signature* for memory accessed by the current transaction. Comparing concurrent transaction signatures allows us to quickly detect conflicting transaction.

1.2.2 Lerna

Lerna [163] is a compiler that runs on the intermediate representation level, which makes it independent of the source code and programming language used. The generated code is a task-based multi-threaded version of the input code.

Unlink HydraVM, Lerna employs static analysis techniques, such as alias analysis and memory dependency analysis, to reduce the overhead of transactional execution, and here the focus is on parallelizing loops. Lerna is not limited to a specific TM implementation, and the integration of any TM algorithm can be done through a well-defined APIs. Additionally, in Lerna the mapping between extracted tasks and transactions is not one-to-one. A transaction can run multiple tasks (tiling), or task can have multiple transactions (partitioning).

Lerna and HydraVM share the idea of exploiting transactional memory, profiling and producing adaptive code. However, as Lerna supports the generation of native executable as output it is not possible to rely on an underlying layer (the virtual machine in HydraVM) to support the adaptive execution. Instead, the program is linked with Lerna runtime library that is able to monitor and modify the key performance parameters of the executor module such as: number of workers threads, the mapping between transactions and tasks, and executing in transaction mode or go sequentially. Both frameworks achieve an average speedup 2.5× on a set of benchmark including applications from JOlden [34], RSTM [2], STAMP [37] and PARSEC [133].

1.2.3 Commitment Order Algorithms

With the experience gained from using HydraVM and Lerna, it is learnt that preserving program order hampers scalability. A thread that completes its execution must either stalls waiting its correct chronological order in the program or proceeds executing more transactions, and consequently increases the lifetime of these pending transactions and makes them subject to conflict with other transactions. Additionally, transactional reads and stores are sandboxed. This prevents any possible *cooperation* between transactions that could still pro-

duce a correct program results. For example, in DOACROSS loops [49] iterations are data or control dependent, however, they can run in parallel with exchanging some data between them.

This dissertation proposes a novel technique [164] for cooperation between concurrent transactions. With this technique, transactions can expose their changes to other transactions without waiting for their chronological commit times. Nevertheless, transactions with the earlier chronological order can abort completed transactions (and cascade abort to any other affected transactions) whenever a conflict exists. Using this technique, the peak gain over the sequential non-instrumented execution in RSTM micro benchmarks is 10× and 16.5× in STAMP.

Furthermore, a set of commit-order algorithms that exploit Intel's Haswell [149] are presented. Haswell is the first mainstream CPU with transactional memory support. The use of HTM support reduces (or eliminates) the transactional execution overhead, which in effect magnifies the parallelization gain.

1.2.4 TM-Friendly Semantics

As mentioned before, the key intuition at the core of HydraVM and Lerna is the use of TM to execute the automatically generated parallel code. However, despite TM's high programmability and generality, its performance is still not yet as good as (or better than) optimized manual implementations of synchronization. Providing high performance in multi-threaded applications before the advent of TM, when thread synchronization was manually done using fine-grained locks and/or lock-free designs, depended upon the specific application semantics. For example, identifying the critical sections and the best number of locks to use are design choices that can be made only after deeply knowing the semantics of the application itself (i.e., what the application does).

A related question that arises in this regard is: *Is there some room for including semantics in TM frameworks without sacrificing their generality?* If the answer is "yes", which is what this dissertation claims and assesses, then we will finally be able to overcome one of the main obstacles that has existed alongside TM since its early stages, and boost its performance accordingly. Motivated by the above question, first, this dissertation identifies a set of semantics that can be included in TM frameworks without impacting the generality of the TM abstraction (*TM-friendly* semantics [162, 161]), and the existing TM APIs are extended to include such semantics. Second, it is shown how to modify STM algorithms to exploit such semantic-based APIs. Results showed speedups of up to 4× over the original algorithms on different applications including micro benchmarks and STAMP.

Finally, embedding those extensions in compiler passes (using GCC) is illustrated so that the application developing experience will not be altered, additionally, such compiler integration permits us to exploit the new TM APIs throughout automatic parallelization frameworks.

The use of the proposed TM extension in automatic parallelization is multi-fold. First, it reduces the aborts by utilizing the embedded application semantics and eliminating some false conflicts. Second, it reduces the number of required TM calls which lower the transaction overhead. Last, it implicitly handles common code patterns such as increments, which automatically eliminates inter-dependencies between transactions (e.g., loop iterations).

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 surveys the techniques for solving the parallelization problem. Past and related efforts is overviewed in Chapter 3. In Chapter 4 and 5, the architecture, transformation, optimization techniques, and the experimental evaluation for the two implementations are detailed: *HydraVM* and *Lerna*.

Chapters 6 and 7 describe the design and the implementation of the commitment order algorithms: Ordered Write Back (OWB) algorithm, and Ordered Undolog (OUL) algorithm. Next, Chapter 8 presents a TM extension for exploiting the application low level semantics.

In Chapter 9, Hardware Transactional Memory (HTM) is exploited for designing two hardware commit-order algorithms: Burning Tickets Algorithm (BTH) and Timeline Flags Algorithm (TFH), and for creating a hybrid version of OWB algorithm, namely OWB-RH. Finally, Chapter 10 concludes the dissertation and proposes future work .

Chapter 2

Background

Parallel computations can be done on multiple levels such as instructions, branch targets, loops, execution traces, and subroutines. Loops have gained a lot of interest as they are by nature a major source of parallelism and usually contain the most processing.

Instruction-Level Parallelization (ILP) is a measure of how many instructions can be run in parallel. ILP is application-specific as it involves reordering the execution of instructions to run in parallel and to utilize the underlying hardware. ILP can be implemented using software (compiler), or hardware (pipelining). Common techniques for supporting ILP are: *out-of-order execution*, where instructions execute in any order that does not violate data dependencies; *register renaming*, which is renaming instruction operands to avoid unnecessary reuse of registers; and *speculative execution*, where an instruction is executed before it should take place according to the serial control flow.

Since on average 20% of instructions are branches, branch prediction was extensively studied to optimize branch execution and to run targets in parallel with the evaluation of branching conditions. Branch predictors are usually implemented in hardware with different variants: Early implementations of SPARC and MIPS used static prediction by always predicting that a conditional jump would not be *taken* and executing the next instruction in parallel to evaluation of the condition. Dynamic prediction is implemented through a state machine that maintains a history of branches (per each branch, or globally). For example, the Intel Pentium processor uses a four-state machine to predict branches. The state machine can be local (per branch instruction) as in the Intel Pentium MMX, Pentium II, and Pentium III; or global (a shared history of all conditional jumps) as in AMD processors and Intel's Pentium M, Core, and Core 2.

Loops can be classified as sequential loops, parallel loops (DOALL), and loops of intermediate parallelism (DOACROSS) [49] (see Figure 2.1). In DOALL loops, iterations are independent and can run in parallel as no data dependencies exist between loops. DOACROSS loops exhibit inter-iteration data dependencies. When data from lower index iterations is used

```
For i = 1 to n {
    Sum = Sum + A[i];
}
```

(a) Sequential Loop

```
For i = 1 to n {
    D[i] = A[i] * B[i] + c
}
```

(b) DOALL Loop

```
For i = 1 to n-1 {
    D[i] = A[i] + B[i]
    C[i] = D[i+1] * B[i]
}
```

(c) Lexically-forward DOACROSS Loop

```
For i = 2 to n {
    D[i] = A[i] + B[i]
    C[i] = D[i-1] * B[i]
}
```

(d) Lexically-backward DOACROSS Loop

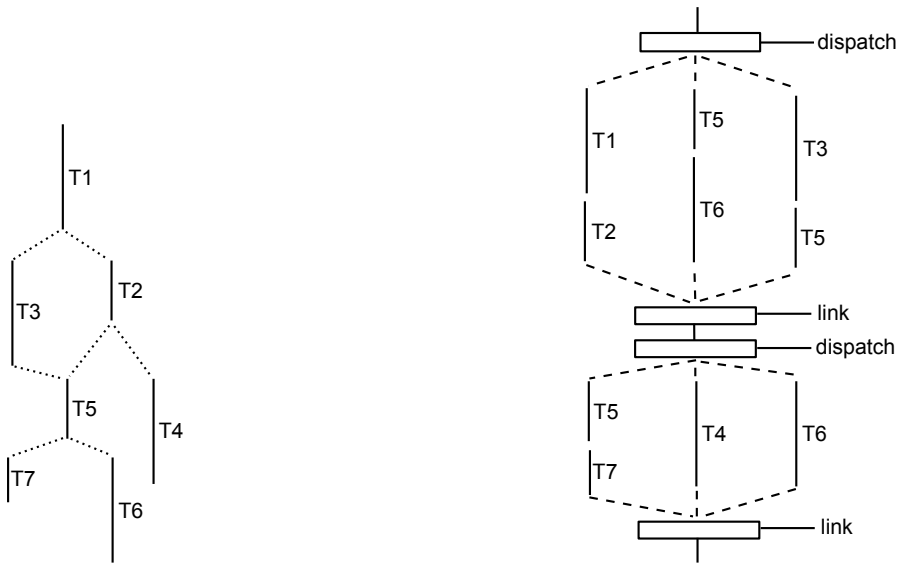
Figure 2.1: Loop classification according to inter-iteration dependencies

by iterations with higher indices, lexically-backward, the processors executing higher index iterations must *wait* for the required calculations from lower index iterations to be evaluated (See Figure 2.1d). In contrast, with lexically-forward loops, lower index iterations access data used at higher indices. Loops can run in parallel without delay if both sets of iterations load their data at their starts.

Traces are defined as hot paths in program execution. Traces can be parts of loops or span multiple iterations and can be parts of individual methods or span multiple methods. A trace is detected dynamically at runtime and is defined as a set of basic blocks (set of instructions ended by a branch or other terminator). Similarly, traces can run in parallel on multiple threads, then are linked together according to their entry and exit points. Figure 2.2 shows an example of traces that run in parallel on three processors.

In contrast to data parallelism (e.g., a loop operating on the same data as in Figure 2.1), *task parallelism* is a form of parallelization wherein tasks or subroutines are distributed over multiple processors. Each task has a different control flow and processes a different set of data; however, they can communicate with each other by passing data between threads. Running different tasks in parallel reduces the overall runtime of a program.

Another classification of parallel techniques is according to user intervention. In *manual* parallelization, the programmer uses programming language constructs to define parallel portions of the code and protects shared data through synchronization primitives. An alternative approach is for the programmer to design and develop the program to run sequentially and use interactive tools that analyze the program (statically or dynamically) to provide the programmer with hints about data dependencies and eligible portions for parallelization. Iteratively, the programmer modifies the code and compares the performance scaling of different threading designs to get the best possible speedup. Clear examples of this *semi-automated* technique are Intel Parallel Advisor [1], and the Paralax compiler [183]. Lastly,



(a) Traces (b) Parallel Traces

Figure 2.2: Running traces over three processors

automatic parallelization aims to parallelize programs without any user intervention.

2.1 Manual Parallelization

Designing a parallel program that runs efficiently in a multi-processor environment is not a trivial procedure as it involves multiple factors and requires a skilled programmer. Maintenance and debugging of parallel programs is a nightmare and incredibly difficult, as it involves racing, invalid shared data accesses, live- and deadlock situations, and non-deterministic scenarios.

Firstly, the programmer must understand the problem that he is trying to solve and the nature of the input data. A problem can be partitioned according to the domain (i.e., input data decomposition), or through functional decomposition (cooperative parallel sub-tasks).

After partitioning the problem, usually there are some kind of communications and shared data accesses between different tasks. Both communications and shared accesses present a challenge (and usually delay) to the parallel program. An application is *embarrassingly parallel* when the communication between its sub-tasks is rare and it does not require a lot of data sharing. The following factors needs to be considered for inter-task communications:

1. *Frequency.* An application exhibits fine-grained parallelism if its sub-tasks communicate frequently, while in coarse-grained parallelism, applications experience less communication. As communication takes place through communication media (e.g., buses),

```
#pragma omp for private(n) shared(total) ordered schedule(dynamic)
for(int n=0; n<100; ++n)
{
    files[n].compress();

    total += get_size(files[n]);

    #pragma omp ordered
    send(files[n]);
}
```

Figure 2.3: OpenMP code snippet

the contention over communication media directly affects overall performance, especially if the media is being used for other purposes (e.g., transferring data from and to processors).

2. *Cost.* The communication cost is determined by: the delay in sending and receiving the information, and the amount of transferred data.

3. *Blocking.* Communication can be either synchronous or asynchronous. Synchronous communications block at the receiver (and sender as well when an acknowledgment is required). On the other hand, asynchronous communications allow both sides to proceed with processing but introduce more complexity to the application design.

Protecting shared data from concurrent access requires a *synchronization* mechanism such as barriers or locks. Synchronization primitives usually introduce bottlenecks and significant overhead to the processing time. Besides, a misuse of locks can cause the application to deadlock (i.e, two or more tasks each waiting for the other to finish), livelock (i.e, tasks are not blocked, but are too busy responding to each other to resume work), or starvation (i.e., greedy set of tasks keep holding the locks for a long time).

Resolving data and control dependencies between tasks is the responsibility of the programmer. A good understanding of the inputs and the underlying algorithm helps in determining independent tasks; however, there are tools that could help in this step [1].

Finally, the programmer should maintain a load balance between the computation resources. For example, static assignment of tasks to processors (e.g., round-robin) is a light technique, but may lead to low utilization; while dynamic assignment of tasks requires monitoring the state of tasks (i.e., start and end times) and handling a shared queue of ready-to-execute tasks.

As an example of parallel programming languages, OpenMP is a compiler extension for the C, C++ and Fortran languages that supports adding parallelism to existing code without

significant changes. OpenMP primarily focuses on parallelizing loops and running iterations in parallel with optional ordering capabilities. Figure 2.3 shows an example of OpenMP parallel code that compresses a hundred files, but sends them in order, and calculates the total size after compression. Programmers can define shared variables (e.g., the non-local variable *total*) and local thread variables (e.g., the loop counter n). Shared variables are transparently protected from concurrent access. A private copy is created for variables that are defined as thread-local (i.e., using the *private* primitive).

2.2 Automatic Parallelization

Past efforts on parallelizing sequential programs can be broadly classified into *speculative* and *non-speculative* techniques. Non-speculative techniques, which are usually compiler-based, exploit loop-level parallelism and differ in the types of data dependencies that they handle (e.g., static arrays, dynamically allocated arrays, pointers) [26, 79, 158, 55].

In speculative techniques, parallel sections of code run speculatively, guarded by a compensating mechanism for handling operations that violate application consistency. The key idea is to provide more concurrency where extra computing resources are available. Speculative techniques can be broadly classified based on

1. what program constructs they use to extract threads (e.g., loops, subroutines, traces, branch targets),
2. whether they are implemented in hardware or software,
3. whether they require source code, and
4. whether they are done online, offline, or both.

Of course, this classification is not mutually exclusive. The execution of speculative code is either *eager* or *predictive*. In eager speculative execution, every path of the code is executed (with the assumption of unlimited resources); however, only the correct value is committed. With predictive execution, selected paths are executed according to prediction heuristics. If there is a misprediction, the execution is rolled back and re-executed.

Among speculative techniques, two appealing primary approaches exist: Thread-Level Speculation (TLS) and Transactional Memory (TM).

2.3 Thread Level Speculation

Thread-Level Speculation (TLS) refers to the execution of out-of-order (unsafe) operations and caching of the results in thread-local storage or buffers (usually via the processor cache).

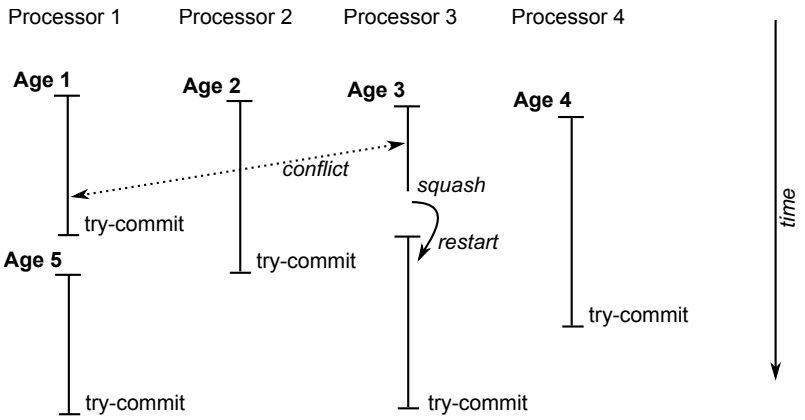


Figure 2.4: Example of thread-level speculation over four processors

TLS assigns an age to each thread according to the earliness of the code it executes. The thread with the earliest age is marked as *safe*.

Speculative threads are subject to buffer overflow. When a thread buffer is full the thread either stalls (till it becomes the lowest age) or is squashed and restarted. An exception is the safe thread (the one executing the earliest code). TLS monitors dependency violations (e.g., through checking cache line requests). For example, a Write-After-Read (WAR) dependency violation occurs when a higher age speculative thread modifies a value before another lower age thread needs to read it. Similarly, when a higher age speculative thread reads an address, then a lower age thread changes the value of this thread, this is another dependency violation named Read-After-Write (RAW). A different type of violation is that caused by control dependencies, where a speculative thread executes unreachable code due to a change in the program control flow. Any dependency violation (data or control) causes the higher-age thread to be squashed and restarted. Figure 2.4 shows an example of TLS using four processors.

TLS is usually implemented through hardware. However, the same concept can be applied using software [105, 158, 55] (with performance issues), especially when low-level operations are not feasible [46, 42] (e.g., with Java-based frameworks). TLS software implementations use a data access signature (or summaries) to detect conflicts between speculative threads. An access signature represents all addresses accessed in the current thread. Conflict is detected by intersecting signatures for partial matches.

2.4 Transactional Memory

Lock-based synchronization is inherently error-prone. Coarse-grained locking, in which a large data structure is protected using a single lock, is simple and easy to use but permits little concurrency. In contrast, with fine-grained locking [123, 95], in which each component

of a data structure (e.g., a bucket of a hash table) is protected by a lock, programmers must acquire necessary and sufficient locks to obtain maximum concurrency without compromising safety. Both these situations are highly prone to programmer errors. In addition, lock-based code is non-composable. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is difficult: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the two tables' methods; if the methods were to export their locks, that will compromise safety. Furthermore, lock-inherent problems such as deadlock, livelock, lock convoying, and priority inversion have not gone away. For these reasons, lock-based concurrent code is difficult to reason about, program, and maintain [89].

Transactional memory (TM) [83] is a promising alternative to lock-based concurrency control. It was proposed as an alternative model for accessing shared memory addresses, without exposing locks in the programming interface, to avoid the drawbacks of locks. With TM, programmers write concurrent code using threads but organize code that reads/writes shared memory addresses as atomic sections. Atomic sections are defined as *transactions* in which reads and writes to shared addresses appear to take effect instantaneously. A transaction maintains a read-set and write-set, and at commit time, checks for conflicts on shared addresses. Two transactions conflict if they access the same address and one access is a write. When that happens, a contention manager [168] resolves the conflict by aborting one of the transactions and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are restarted, often immediately. Thus, a transaction ends by either committing (i.e., its operations take effect) or by aborting (i.e., its operations have no effect). In addition to a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking implementations [59, 33] and is composable [84]. Multiprocessor TM has been proposed in hardware (HTM), in software (STM), and in hardware/software combinations (HyTM).

Transactional Memory algorithms differ according to their design choices. An algorithm can apply its changes directly to memory and maintain an undo-log for restoration to a consistent state upon failure. Such an optimistic approach fits the situation where conflicts are rare. In contrast, an algorithm can use a private thread-local buffer to keep its changes invisible from other transactions. At commit, the local changes are merged into main memory. Another orthogonal design choice is the time of conflict detection. Transactions may acquire access (lock) on their write-sets at encounter time or during the commit phase.

Figure 2.5 shows some example transactional code. Atomic sections are executed as transactions. Thus, the possible values of A and B are either 42 and 42, or 22 and 11, respectively. An inconsistent view of a member (e.g., A=20 and B=10), due to atomicity violation or interleaved execution, causes one of the transactions to abort, rollback, and then re-execute.

Motivated by TM's advantages, several recent efforts have exploited TM for automatic parallelization. In particular, trace-based automatic/semi-automatic parallelization is explored in [31, 32, 39, 58], which use HTM to handle dependencies. [144] parallelizes loops with

A = 10, B = 20;

THREAD A

```
atomic{
    B = B + 1;
    A = B * 2;
}
....
```

THREAD B

```
atomic{
    B = A;
}
....
....
```

Figure 2.5: An example of transactional code using atomic TM constructs

dependencies using thread pipelines, wherein multiple parallel thread pipelines run concurrently. [120] parallelizes loops by running them as transactions, with STM preserving the program order. [173] parallelizes loops by running a non-speculative "lead" thread, while other threads run other iterations speculatively, with STM managing dependencies.

2.4.1 NOrec

The No Ownership Records Algorithm (NOrec) [53] is a lazy software transactional memory algorithm that uses a minimal amount of metadata for accessed memory addresses. Unlike other STM algorithms, NOrec does not associate ownership records (orecs) to maintain its write-set. Instead, it uses a value-based validation during commit time to make sure the read values still have the same values the transaction already used during its execution.

Transactions use a write-buffer to store their updates; the implementation of the write-buffer is a linear-probed hash table with versioned buckets to support O(1) clearing (when transaction descriptors are reused). NOrec uses a *lazy* locking mechanism, which means written addresses are not locked until commit time. This approach reduces locking time for the accessed memory locations, which allows readers to proceed without writer interference.

NOrec employs a single global sequence lock. Whenever a transaction commits, the global lock is acquired and is incremented. This assumption limits the system to having a single committer transaction at a time. The commit procedure starts by incrementing the sequence lock atomically. Failing to increment the lock means another writer transaction is trying to commit, so the current transaction needs to validate its read-set and wait for the other transaction to finish the commit. Upon successful increment, the transaction acquires locks on its write-set, exposes the changes to memory, and releases the locks.

Another drawback of the algorithm is that before each read, it must validate the whole read-set. This is required to maintain opacity [75] – a TM feature which mandates that even invalid transactions must always read a consistent view of memory. NOrec tries to avoid unneeded validation by doing it whenever the global sequence got changed (i.e., when

a transaction commit takes place).

NOrec is the default TM algorithm for Lerna. In Chapter 5, we describe our variant of the algorithm that preserves ordering between concurrent transactions.

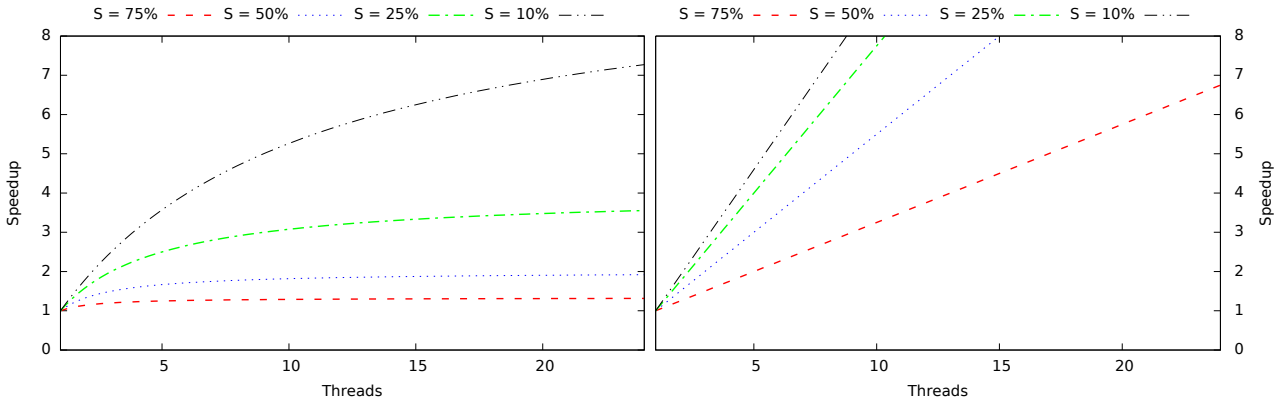
2.4.2 TinySTM

TinySTM [69] is a lightweight, lock-based STM algorithm. It uses a single-version word-based variant of the LSA [150] algorithm. Similar to other word-based locking algorithms [53, 59], TinySTM relies upon a shared array of locks that cover all memory addresses. Multiple addresses are covered by the same lock, and each lock uses a single bit for state (i.e., locked or not) with the remaining bits as a version number. This version number indicates the timestamp of the last transaction that wrote to any of the addresses covered by this lock. As a lock covers a portion of the address space, false conflicts could occur when concurrent transactions access adjacent addresses.

Unlike NOrec, TinySTM uses Encounter-Time Locking (ETL); transactions acquire locks during execution. The benefit of ETL is twofold: conflicts are discovered at an early time, which avoids wasting processing time executing doomed transactions; and the handling of read-after-write situations is simplified. The algorithm uses a time-based design [59, 150] by employing a shared counter as a clock. Update transactions acquire a new timestamp on commit, validate their read-sets, then store the timestamp to the versioned locks of their write-sets.

TinySTM is proposed with two strategies for memory access: write-through and write-back. Each has its advantages and limitations. Using the write-back strategy, updates are kept in a local transaction write-buffer until commit time, while in write-through updates go to main memory and the old values are stored in an undo log. With write-through, transactions have lower commit-time overhead and faster read-after-write/write-after-write handling. However, aborts are costly as they require restoring the old values of written addresses. On the other hand, in write-back, the abort procedure simply discards the read and write sets, but commit requires validating the read-set and moving the write-set values from the local write-buffer to main memory.

The use of ETL is interesting to our work as it enables early detection of conflicting transactions, which saves processing cycles. Additionally, the write-through strategy is perfect for low-contention workloads as it involves lightweight commits that could lead to performance comparable to sequential execution.



(a) Amdahl's Law (b) Gustafson's Law

Figure 2.6: Maximum Speedup according to Amdahl's Law and Gustafson's Laws

2.5 Parallelism Limits and Costs

Amdahl's law is a formula that defines the upper bound on expected speedup to an overall system when only part of the system is improved. Let S be the percentage of the program's serial portion of the code (i.e., not included in the parallelism). When executing the program using n threads, then the expected execution time is

$$Time(n) = Time(1) * (S + 1/n * (1 - S))$$

Therefore, the maximum speedup is given by the following formula

$$Speedup(n) = 1/(S + 1/n * (1 - S)) = n/(1 + S * (n - 1))$$

Figure 2.6a shows the maximum possible speedup for different percentages of the sequential portion of the code with different numbers of threads. With only 10% of the code being sequential, the maximum speedup using 20 threads is around 7× only. Thus, at some point adding an additional processor to the system will add less speedup than the previous one, as the total speedup heads toward the limit of $1/(1 - n)$. However, here we assume a fixed size of the input. Usually, adding more processors permits solving larger problems (i.e., increasing the input size), consequently increasing the parallel portion of the program. This fact is captured by Gustafson's Law, which states that computations involving arbitrarily large data sets can be efficiently parallelized. Accordingly, the speedup can be defined by the following formula (See Figure 2.6b)

$$Speedup(n) = n - S * (n - 1)$$

In general, parallel applications are much more complex than sequential ones. The complexity appears in every aspect of the development cycle including: design, development, debugging, tuning, and maintenance. Add to that the hardware requirements for running a parallel system. The return value per added processor is not guaranteed to be reflected in the overall performance. On the contrary, sometimes splitting the workload over even more threads increases rather than decreases the amount of time required to finish. This is known as *parallel slowdown*.

Chapter 3

Past & Related Work

3.1 Transactional Memory

The classical solution for handling shared memory during concurrent access is to protect shared addresses with locks [11, 98]. However, locks have many drawbacks including deadlock, livelock, lock-convoying, priority inversion, non-composability, and the overhead of lock management.

TM, proposed by Herlihy and Moss [93], is an alternative approach for shared memory access with a simpler programming model. Memory transactions are similar to database transactions: a memory transaction is a self-maintained entity that guarantees atomicity (all or none), isolation (local changes are hidden till commit), and consistency (linearizable execution). TM has gained significant research interest including that for Software Transactional Memory (STM) [170, 127, 84, 82, 91, 92, 116], Hardware Transactional Memory (HTM) [93, 81, 10, 28, 128], and Hybrid Transactional Memory (HyTM) [19, 54, 129, 104]. STM has relatively larger overhead due to transaction management and architecture-independence. HTM has the lowest overhead but assumes architecture specializations. HyTM seeks to combine the best of HTM and STM.

STM can be broadly classified as static or dynamic. In static STM [170], all accessed addresses are defined in advance, while dynamic STM [91, 92, 116] relaxes that restriction. The dominant trend in STM designs is to implement the single-writer/multiple-reader pattern, either using locks [60, 59] or obstruction-free (i.e., a single thread executed in isolation will complete its operation with a bounded number of steps) techniques [150, 91], though a few implementations allow multiple writers to proceed under certain conditions [151]. In fact, it is shown in [67] that obstruction-freedom is not an important property and results in less efficient STM implementations than lock-based ones.

Another orthogonal TM property is address acquisition time: pessimistic approaches acquire

addresses at encounter time [67, 33], while optimistic approaches do so at commit time [60, 59]. Optimistic address acquisitions generally provide better concurrency with an acceptable number of conflicts [59]. STM implementations also rely on write-buffer [114, 116, 91] or undo-log [129] approaches to ensure a consistent view of memory. In the write-buffer approach, address modifications are written to a local buffer and take effect at commit time. In the undo-log method, writes directly change the memory and the old values are kept in a separate log to be retrieved at abort.

Nesting (or composability) is an important feature for transactions as it allows partial rollback and introduces semantics between the parent transaction and its enclosed ones. Earlier TM implementations did not support nesting or simply flattened nested transactions into a single top-level transaction. Harris *et. al.* [84] argue that closed nested transactions, supporting partial rollback, are important to implementing *composable* transactions, and presented an $orElse$ construct that relies upon closed nesting. In [6], Adl-Tabatabai *et. al.* presented an STM that provides both nested atomic regions and $orElse$, and introduced the notion of mementos to support efficient partial rollback. Recently, a number of researchers have proposed the use of open nesting. Moss described the use of open nesting to implement highly concurrent data structures in a transactional setting [130]. In contrast to a database setting, the different levels of nesting are not well-defined; thus, different levels may conflict. For example, a parent and child transaction may both access the same memory location and conflict.

3.2 Parallelization

Parallelization has been widely explored using different levels of programmer intervention. Manual parallelization techniques rely on the programmer for analysis and design phases but assist in the implementation and performance tuning. Open Multi-Processing (OpenMP) [51] defines an API that supports shared memory multiprocessing programming in C, C++, and Fortran. The programmer uses OpenMP directives to define parallel sections of the code and the execution semantics (e.g., ordering). Synchronization of shared access is handled through directives that define the scope of access (i.e., shared or private). OpenMP transparently and efficiently handles low-level operations such as locking, scheduling and thread stalling. Message Passing Interface (MPI) [171] is a message-based, language-independent communications protocol used for parallel computing. MPI offers basic concepts for messaging between parallel processes such as: grouping and partitioning of processes, various types of communication (i.e., point-to-point, broadcasting or reduce), interchanged/custom data types, and synchronization (global, pairwise, and remote locks). NVIDIA introduced CUDA [138] as a parallel programming and computing platform for its graphics processing units (GPUs); this enables programmers to access a GPU's virtual instruction set and memory and use it for general purpose processing (not exclusively graphics computation). GPUs rely on using *many* concurrent slow threads rather than using limited count of cores with

high speed (as in CPUs), which makes GPUs suitable for data-parallel computations.

An alternative approach, semi-automatic parallelization, provides the programmer with hints and design decisions that help him write code that is more eligible for parallelization or detects data dependencies and shared accesses during the design phase when they are less expensive to fix. Paralax [183] is a compiler framework for extracting parallel code using static analysis. Programmers use annotations to assist the compiler in finding parallel sections. DiscoPoP [110] and Kremlin [71] are runtime tools that discover and present ranked suggestions for parallelization opportunities. Intel Parallel Advisor (Advisor XE) [1] is a shared-memory threading design and prototyping tool. Advisor XE helps programmers detect parallel sections and compare performance using different threading models. Additionally, it finds data dependencies and enables eliminating them, if possible.

Automatic parallelization aims to produce best-effort performance without any programmer intervention; this relieves programmers from designing and writing complex parallel applications; besides that, it is highly useful for legacy code. Polaris [68], one of the earliest parallelizing compiler implementations, proposes a source-to-source transformation that generates a parallel version of the input program (Fortran). Another example is the Standford SUIF compiler [79]; it determines parallelizable loops using a set of data dependency techniques such as data dependence analysis, scalar privatization analysis, and reduction recognization. SUIF optimizes cache usage by ensuring that processors reuse the same data and defragment shared addresses.

Parallelization techniques can be classified according to their granularity. Significant efforts have focused on enhancing fine-grained parallelism such as that for: nested loops [30, 146, 49], regular/irregular array accesses [159], and scientific computations [192] (e.g., dense/sparse matrix calculations). Sohi *et. al.* [172] increase instruction-level parallelism by dividing tasks for speculative execution amongst functional units; [154] does so with a trace-based processor. Nikolov *et. al.* [137] use a hybrid processor/SoC architecture to exploit nested loop parallelism, while [9] uses postdominance information to partition tasks on a multithreaded architecture. However, fine-grained parallelism is not sufficient for exploitation of CMP parallelism. Coarse-grained parallelism focuses on parallelizing tasks as units of work. In [179], the programmer manually annotates concurrent and synchronized code blocks in C programs and then uses those annotations for runtime parallelization. Gupta *et. al.* [78] and Rugina *et. al.* [158] do compile-time analysis to exploit parallelism in array-based, divide-and-conquer programs.

3.3 Optimistic Concurrency

Optimistic concurrency techniques, such as thread-level speculation (TLS) and Transactional Memory (TM), have been proposed as a way of extracting parallelism from legacy code. Both techniques split an application into sections using hardware or a compiler and run

them speculatively on concurrent threads. A thread may buffer its state or expose it and use a compensating procedure. At some point, the executed code may become safe and the code proceeds as if it were executed sequentially. Otherwise, the code's changes are reverted and execution is restarted. Some efforts combined TLS and TM through a unified model [18, 145, 144] to get the best of the two techniques.

3.3.1 Thread-Level Speculation

Automatic parallelization for thread-level speculation (TLS) hardware has been extensively studied, with focus largely on loops [146, 181, 65, 111, 176, 43, 45, 142, 186]. Loop parallelization using TLS has been proposed in both hardware [141] and software [146, 22]. The LRPD Test [146] determines if a loop has any cross-iteration dependencies (i.e., DOALL loop) and runs it speculatively; at runtime, a validation step is performed to check if the accessed data is affected by any unpredictable control flow. Saltz and Mirchandane [167] parallelize DOACROSS loops by assigning iterations to processors in a wrapped manner. To prevent data dependency violations, processors have to stall till the correct values are produced. Polychronopoulos [140] proposes running the maximal set of continuous iterations with no dependencies concurrently; consequently, this method does not fully utilize all processors. An alternative approach is to remove dependencies between the iterations. Krothapalli *et. al.* [103] proposed a runtime method to remove anti Write-After-Read (WAR) and Write-After-Write (WAW) dependencies. This method helps to remove dependencies caused by reusing memory/registers, but does not remove computation dependencies.

Prabhu *et. al.* [141] presented some guidelines for programmers to manually parallelize code using TLS. In their work, a source-to-source transformation is performed in order to run the loops speculatively, while TLS hardware detects any data dependency violations and acts accordingly (i.e., squashing the threads and restarting the iteration). Zilles *et. al.* [191] introduced an execution paradigm called Master/Slave Speculative Parallelization (MSSP). In MSSP, a master processor executes an approximate version of the program, based on common-case execution, to compute selected values that the full program's execution is expected to compute. The masters results are checked by slave processors that execute the original program.

Automatic and semi-automatic parallelization without TLS hardware have also been explored [105, 158, 55, 46, 42]. In [144], Raman *et. al.* proposed a software approach that generalizes existing software TLS memory systems to support speculative pipelining schemes and efficiently tunes them for loop parallelization. Jade [105] is a programming language that supports coarse-grained concurrency and exploits a software TLS technique. Using Jade, programmers augment the code with data dependency information and the compiler uses this to determine which operations can be executed concurrently. Deutsch [55] analyzes symbolic access paths for interprocedural may-alias analysis with the goal of exploiting parallelism. In [46], Choi *et. al.* present escape analysis for Java programs for determining

object lifetimes for concurrency enhancement.

3.3.2 Parallelization using Transactional Memory

Tobias *et. al.* [66] proposed an epoch-based speculative execution of parallel traces using hardware transactional memory (HTM). Parallel sections are identified at runtime based on binary code. The conservative nature of the design does not utilize all cores; besides, relying on techniques only at runtime for parallelization introduces nonnegligible overhead to the framework. Similarly, DeVuyst *et. al.* [56] used HTM to optimistically run parallel sections, which are detected using special hardware. Sambamba [178] showed that static optimization at compile-time does not exploit all possible parallelism. Similar to our work, it used Software Transactional Memory (STM) with an adaptive runtime approach for executing parallel sections. It relies on user input for defining parallel sections. Gonzalez *et. al.* [73] proposed a user API for defining parallel sections and ordering semantics. Based on user input, STM is used to handle concurrent sections. In contrast, HydraVM does not require special hardware and is fully automated, with optional user interaction for improving speedup.

The study at [184] classified applications into sequential, optimistically parallel, or truly parallel; and classified tasks into ordered (speculative iterations of loop) and unordered (critical sections). It introduced a model that captures data and inter-dependencies. For a set of benchmarks [37, 15, 135, 41], the study showed important features for each, like size of read and write sets, dependency density, and size of parallel sections.

Mehrara *et. al.* [120] present *STMlite*: a lightweight STM implementation customized to facilitate profile-guided automatic loop parallelization. In this scheme, loop iterations are divided into chunks and distributed over available cores. An outer loop is generated around the original loop body to manage parallel execution between different chunks.

In MTX, a transaction runs under more than one thread. Vachharajani *et. al.* [182] presented a hardware memory system that supports MTXs. Their system changes the hardware cache coherence protocol to buffer speculative states and recover from mis-speculation. Software Multi-threaded Transactions (SMTX) are used to handle memory accesses for speculated threads. Both SMTX and STMlite use a centralized transaction commit manager and conflict detection that is decoupled from the main execution.

3.4 Comparison with existing work

Most of the methodologies, tools and languages for parallelizing programs target scientific and data-parallel, computation-intensive applications, where actual data sharing is very limited and the dataset is precisely analyzed by the compiler and partitioned so that parallel computation is possible. Examples of those approaches include [134, 153, 111, 94].

Despite the flexibility and ease of use of TM, the primary drawback of using it for parallelization is the significant overhead of code execution and validation [38, 166]. Previous work [73, 178] relied primarily on the programmer for specification of parallel sections and definition of ordering semantics. The key weaknesses with programmer reliance are:

1. it requires a full understanding of the software (e.g., the algorithm implementation and input characteristics) and mandates the existence of the source code;
2. it does not take into account TM characteristics and factors of overhead; and
3. it uses TM as a black box utility for preserving data integrity.

In contrast, the solutions proposed in this thesis target common programs and analyze the code in its intermediate representation without the need for the original source code. This makes us independent of the language in which the code was written and does not enforce exposure of the source code to our framework, especially when it is not available.

Additionally, in Lerna, we employ alias analysis and propose two novel techniques: *high-priority transactions* and *transactional increment*, for reducing the read-set size and validation overhead and eliminating some reasons of transactional conflicts. As far as we know, this is the first study that attempts to reduce transactional overhead based on program characteristics.

HydraVM and MSSP [191] share the same concept of using execution traces (not just loops such as [120, 173]). However, MSSP uses superblock [96] executions for validating the main execution. In contrast, HydraVM splits execution equally on all threads and uses STM for handling concurrent memory accesses. Perhaps the closest to our proposed work are [32] and [66]. Our work differs from [32] and [66] in the following ways. First, unlike [66], we propose STM for concurrency control, which does not need any hardware transactional support. Second, [32] is restricted to recursive programs, whereas we allow arbitrary programs. Third, [32] does not automatically infer transactions; rather, entire work performed in tasks (of traces) is packaged as transactions. In contrast, we propose compile- and runtime program analysis techniques that identify traces, which are executed as transactions.

Our work is fundamentally different from past STM-based parallelization works in that we benefit from static analysis for reducing transactional overheads and automatically identify parallel sections (i.e., traces or loops) by compile- and runtime program analysis techniques, which are then executed as transactions. Additionally, our work targets arbitrary programs (not just recursive ones as [32] does), is entirely software-based (unlike [32, 66, 56, 182]), and does not require program source code. Thus, our proposed work of STM-based parallelization (with the consequent advantages of STM's concurrency control, completely software-based, and no need for program sources), has never been done before.

Chapter 4

HydraVM

In this chapter, we present a virtual machine, called HydraVM, that automatically extracts parallelism from legacy sequential code (at the bytecode level) through a set of techniques including online and offline code profiling, data dependency analysis, and execution analysis. HydraVM is built by extending the Jikes RVM [12] and modifying its baseline compiler, and exploits software transactional memory to manage concurrent and out-of-order memory accesses.

HydraVM targets extracting parallel *jobs* in the form of code traces. This approach is different from loop parallelization [30], because a trace is equivalent to an execution path which can be a portion of a loop, or spans loops and method calls. Traces were invented in [16] as part of HP's Dynamo optimizer, which optimizes native program binary at runtime using a trace cache.

To handle potential memory conflicts, we develop ByteSTM, which is a VM-level STM implementation. In order to preserve original semantics, ByteSTM suspends completed transactions till their valid commit times are reached. Aborted transactions discard their changes and are either terminated (i.e., a program flow violation or a misprediction) or re-executed (i.e., to resolve a data-dependency conflict).

We experimentally evaluated HydraVM on a set of benchmark applications, including a subset of the JOlden benchmark suite [34]. Our results reveal speedup of up to 5× over the sequential code.

4.1 Program Reconstruction

The program can be represented as a set of *basic blocks*, where each basic block is a sequence of non-branching instructions that ends either with a branch instruction (conditional or non-conditional) or a return. Thus, any program can be represented by a directed graph in which nodes represent basic blocks and edges represent the program control flow – i.e., Control Flow Graph (CFG). Basic blocks can be determined at compile-time. However, our main goal is to determine the context and frequency of reachability of the basic blocks – i.e., when the code is revisited through execution.

Basic blocks can be grouped according to their runtime behavior and execution paths. A *Trace* is a set of connected basic blocks at the CFG, and it represents an execution path (See Figure 4.1a). A *Trace* contains one or more conditional branches that may transfer the control out of the trace boundaries, namely *exits*. Exists transfer control to the program or to another trace. It is possible for a *trace* to have more than one *entry*, and a trace with a single entry is named a *Superblock* [96].

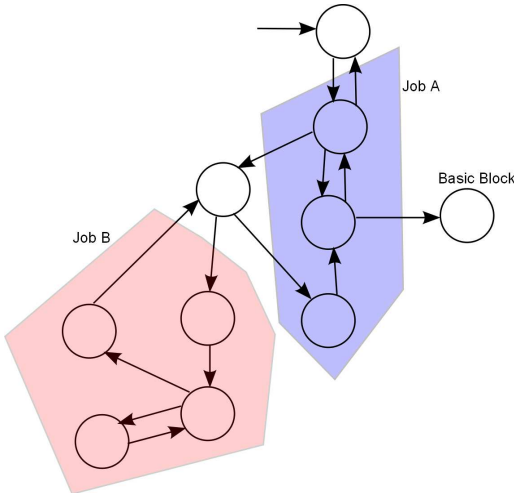
Our basic idea is to optimistically split code into parallel traces. For each trace, we create a synthetic method (See Figure 4.1b) that:

- Contains the code of the trace.
- Receives its entry point and any variables accessed by the trace code as input parameters.
- Returns the *exit* point of the trace (i.e., the point where the function returns).

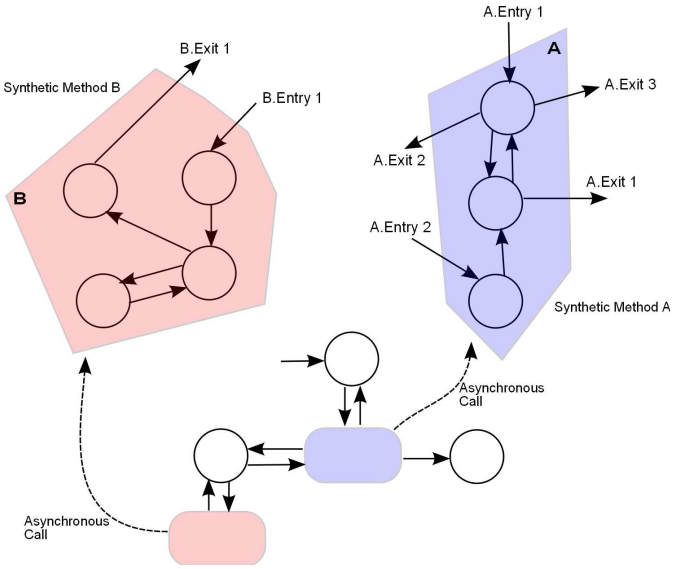
Synthetic methods are executed in separate threads as memory transactions, and a TM library is used for managing the contention. We name a call to the synthetic methods with a specific set of inputs as a *job*. Multiple jobs can execute the same code (trace), but with different inputs. As concurrent jobs can access and modify the same memory location, it is required to protect memory against invalid accesses. To do so, we employ TM to organize access to memory and to preserve memory consistency. Each transaction is mapped to a subset of the job code or spans multiple jobs.

While executing, each transaction operates on a private copy of the accessed memory. Upon a *successful* completion of the transaction, all modified variables are exposed to the main memory. We define a *successful execution* of an invoked job as an execution that satisfies the following two conditions:

- It is reachable by future executions of the program; and
- It does not cause a memory conflict with any other job having an older chronological order.



(a) Control Flow Graph with two Traces



(b) Transformed Program

Figure 4.1: Program Reconstruction as Jobs

As we will detail in Section 4.2, any execution of a parallel program produced after our transformations is made of a sequence of jobs committed after a successful execution.

In a nutshell, the parallelization targets those blocks of code that are prone to be parallelized and uses the TM abstraction to mark them. Such TM-style transactions are then automatically instrumented by us to make the parallel execution correct (i.e., equivalent to the execution of the original serial application) even in presence of data-conflicts (e.g., the case of two iterations of one loop activated in parallel and modifying the same part of a shared data structure). Clearly the presence of more conflicts leads to less parallelism and thus poor performance.

4.2 Transactional Execution

TM encapsulates optimism: a transaction maintains its read-set and write-set, i.e., the objects read and written during the execution, and at commit time checks for conflicts on shared objects. Two transactions conflict if they access the same object, and one access is a write. When this happens, a contention manager [168] solves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, often immediately. Thus, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect).

The atomicity of transactions is mandatory as it guarantees the consistency of the code, even after its refactoring to run in parallel. However, if no additional care is taken, transactions run and commit independently of each other, and that could revert the chronological order of the program, which must be preserved to avoid incorrect executions.

In our model, jobs run (which contain transactions) speculatively, but a transaction, in general, is allowed to commit whenever it finishes. This property is desirable to increase thread utilization and avoid fruitless stalls, but it can lead to transactions corresponding to unreachable code (e.g., a break condition that changes the execution flow), and transactions executing code with earlier chronological order may read future values from committed transactions that are corresponding to code with later chronological order. The following example illustrates this situation.

Consider the example in Figure 4.2, where three jobs A, B, and C are assigned to different threads T_A, T_B, and T_C and execute as three transactions t_A, t_B, and t_C, respectively. Job A can have B or C as its successor, and that cannot be determined until runtime. According to the parallel execution in Figure 4.2(b), T_C will finish execution before others. However, t_C will not commit until t_A or t_B completes successfully. This requires that every transaction must notify the STM to permit its successor to commit.

Now, let t_A conflict with t_B because of unexpected memory access. STM will favor the older transaction in the original execution and abort t_B, and will discard its local changes.

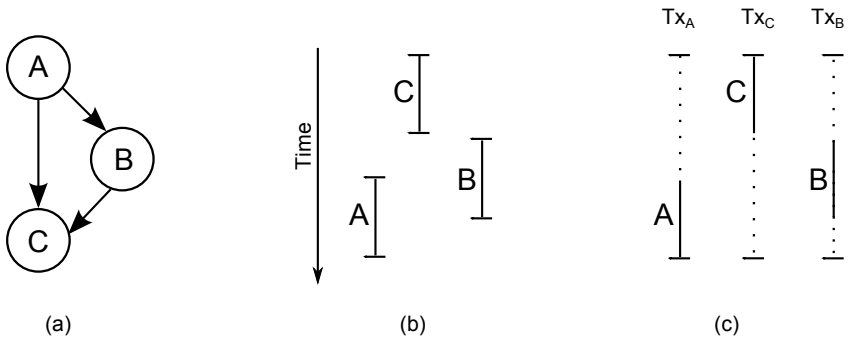


Figure 4.2: Parallel execution pitfalls: (a) Control Flow Graph, (b) Possible parallel execution scenario, and (c) Managed TM execution.

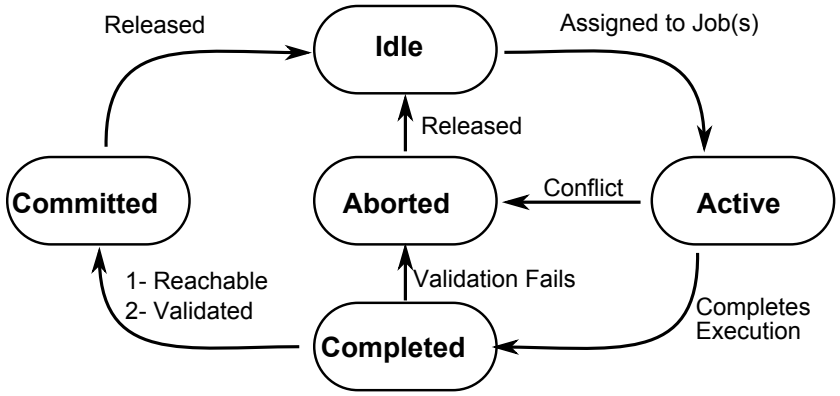


Figure 4.3: Transaction States

Later, t_B will be re-executed. A problem arises if t_A and t_C wrongly and unexpectedly access the same memory location. Under Figure 4.2(b)'s parallel execution scenario, this will not be detected as a transactional conflict (T_C finishes before T_A). To handle this scenario, we extend the lifetime of transactions to the earliest transaction starting time. When a transaction must wait for its predecessor to commit, its lifetime is extended till the end of its predecessor. Figure 4.2(c) shows the execution from the our managed TM perspective.

Although these scenarios are admissible under generic concurrency controls (where the order of transactions is not enforced), it clearly violates the logic of the program. To resolve this situation, program order is maintained by deferring the commit of transactions that complete early till their valid execution time.

Motivated by that, we propose an ordered transactional execution model based on the original program's chronological order. Transactions execution works as follows. Transactions have five states: *idle*, *active*, *completed*, *committed*, and *aborted*. Initially, a transaction is idle because it is still in the transactional pool waiting to be attached to a job to dispatch. Each transaction has an *age* identifier that defines its chronological order in the program. A transaction becomes active when it is attached to a thread and starts its execution. When

a transaction finishes the execution, it becomes completed. That means that the transaction is ready to commit, and it completed its execution without conflicting with any other transaction. A transaction in this state still holds all locks on the written addresses. Finally, the transaction is committed when it becomes reachable from its predecessor transaction. Decoupling *completed* and *committed* states, permits threads to process next transactions without the need to wait for the transaction valid execution time.

4.3 Jikes RVM

Jikes Research Virtual Machine (Jikes RVM) is an open source implementation Java Virtual Machine (JVM). Jikes RVM has a flexible and modular design that support prototyping, testbed and doing the experimental analysis. Unlike most other JVM implementations, which is usually written in native code (e.g., C, C++), Jikes is written in Java. This characteristic provides portability, object-oriented design, and integration with the running applications.

Jikes RVM is divided to the following components:

- *Core Runtime Services*: This component is responsible for managing the execution of running applications, which includes the following:
  - Thread creation, management and scheduling.
  - Loading classes definitions and triggering compiler.
  - Handling calls to Java Native Interface (JNI) methods
  - Exception handling and traps
- *Magic*: This is a mechanism for handling low-level system-programing operations such as: raw memory access, uninterruptible codes, and unboxed types. Unlike all other components, this module is not written in pure Java, as it uses machine code to provide this functionality.
- *Compilers*: it reads bytecode and generates an efficient machine code that is executable for the current platform
- *The Memory Manager Toolkit (MMTK)* handles memory allocation and garbage collection.
- *Adaptive Optimization System (AOS)* allows online feedback-directed optimizations. It is responsible for profiling an executing application and triggers the optimizing compiler to improve its performance.

4.4 System Architecture

In HydraVM, we extend the AOS [12] architecture to enable parallelization of input programs, and dynamically refine parallelized sections based on execution. Figure 4.4 shows HydraVM's architecture, which contains six components:

- Profiler: performs static analysis and adds additional instructions to monitor data access and execution flow.
- Inspector: monitors program execution at runtime and produces profiling data.
- Optimization Compiler: recompiles bytecode at runtime to improve performance and triggers reloading classes definitions.
- Knowledge Repository: a store for profiling data and execution statistics.
- Builder: uses profiling data to reconstruct the program as multi-threaded code, and tunes execution according to data access conflicts.
- TM Manager: handles transactional concurrency control to guarantee safe memory and preserves execution order.

HydraVM works in three phases. The first phase focuses on detecting parallel patterns in the code, by injecting the code with hooks, monitoring code execution, and determining memory access and execution patterns. This may lead to slower code execution due to inspection overhead. *Profiler* is active only during this phase. It analyzes the bytecode and instruments it with additional instructions. *Inspector* collects information from generated instructions and stores it in the Knowledge Repository.

The second phase starts after collecting enough information in the *Knowledge Repository* about which blocks were executed and how they access memory. The *Builder* component uses this information to split the code into traces, which can be executed in parallel. The new version of the code is generated and is compiled by the *Recompiler* component. The *TM Manager* manages memory access of the execution of the parallel version, and organizes transactions commit according to the original execution order. The manager collects profiling data including commit rate and conflicting threads.

The last phase is tuning the reconstructed program based on thread behavior (i.e., conflict rate). The Builder evaluates the previous reconstruction of traces by splitting or merging some of them, and reassigning them to threads. The last two phases work in an alternative way till the end of program execution, as the second phase represents a feedback to the third one.

HydraVM supports two modes: *online* and *offline*. In the online mode, we assume that program execution is long enough to capture parallel execution patterns. Otherwise, the

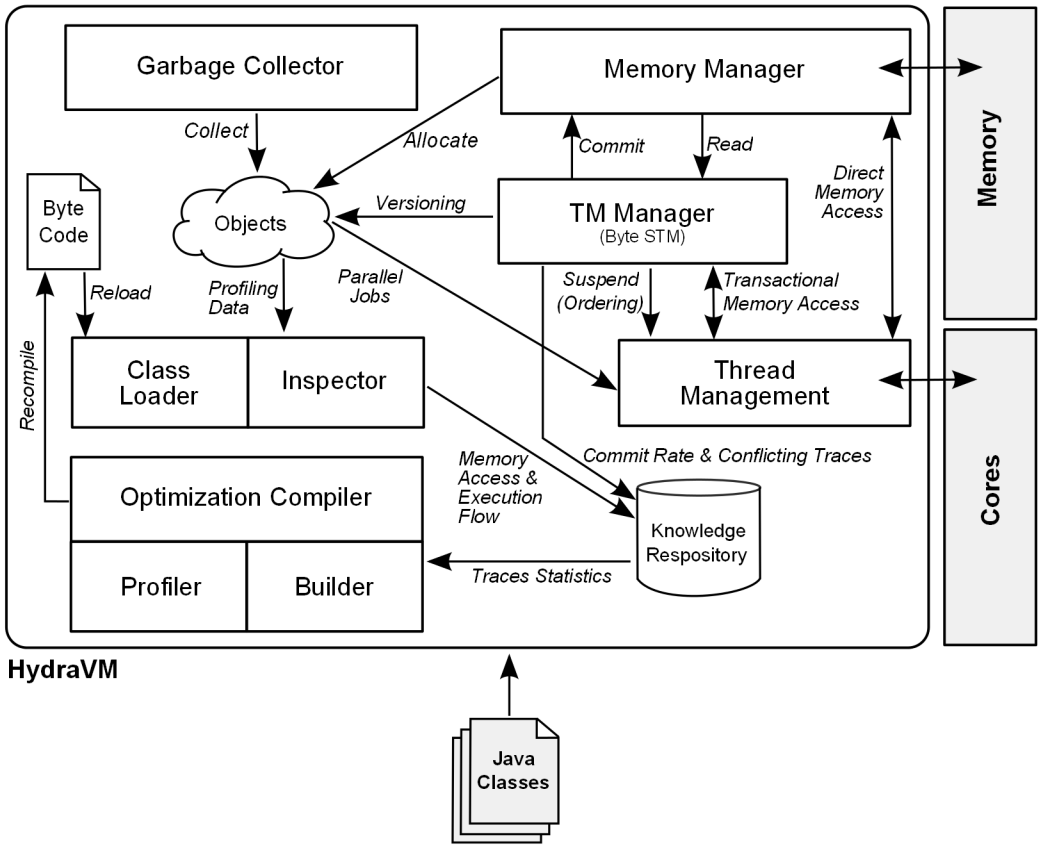


Figure 4.4: HydraVM Architecture

first phase can be done in a separate pre-execution phase, which can be classified as offline mode.

We now describe each of HydraVM's components.

4.4.1 Bytecode Profiling

To collect this information, we modify Jikes RVM's baseline compiler to insert additional instructions (in the program bytecode) at the edges of selected basic blocks (e.g., branching, conditional, return statements) that detect whenever a basic block is reached. Additionally, we insert instructions into the bytecode to:

- Statically detect the set of variables accessed by the basic blocks, and
- Mark basic blocks with irrevocable calls (e.g., input/output operations), as they need special handling in program reconstruction.

This code modification does not affect the behavior of the original program. We call this version of the modified program, *profiled bytecode*.

4.4.2 Trace detection

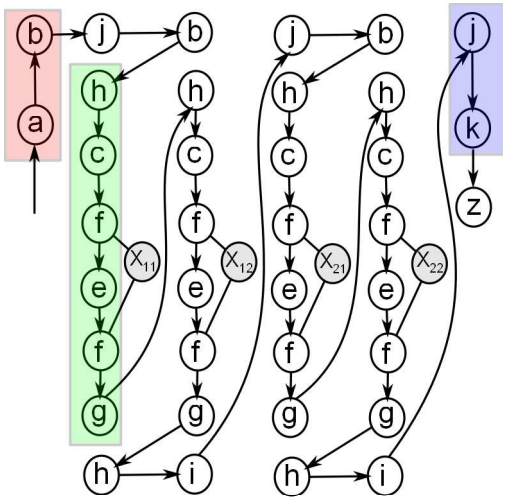
With the profiled bytecode, we can view the program execution as a graph with basic blocks and variables represented as nodes, and the execution flow as edges. A basic block that is visited more than once during execution will be represented by a *different* node each time (See Figure 4.5b). The benefits of execution graph are multifold:

- Hot-spot portions of the code can be identified by examining the graph's hot paths,
- Static data dependencies between blocks can be determined, and
- Parallel execution patterns of the program can be identified.

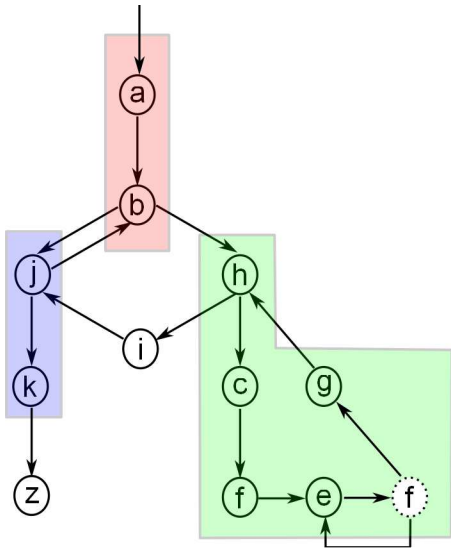
To determine traces, we use a string factorization technique: each basic block is represented by a character that acts like a unique ID for that block. Now, an execution of a program can be represented as a string. For example, Figure 4.5a shows a matrix multiplication code snippet. An execution of this code for a 2x2 matrix can be represented with the execution graph shown at Figure 4.5b, or as the string $abjbhcfefghcfefghijbhcfefghcfefghijk$. We factorize this string into its basic components using a variant of Main's algorithm [113]. The factorization converts the matrix multiplication string into $ab(jb(hcfefg)^2hi)^2jk$. Using this representation, combined with grouping blocks that access the same memory locations, we divide the code into multiple sets of basic blocks, namely *traces* (See Figure 4.5c). In our example, we detected three traces:

```
for (Integer i = 0; i < DIMx; i++)
    for (Integer j = 0; j < DIMx; j++)
        for (Integer k = 0; k < DIMy; k++)
            X[i][j] += A[i][k] * B[k][j];
```

(a) Matrix Multiplication Code



(b) 2x2 Matrix Multiplication Execution Graph with Traces



(c) Control Flow Graph with Traces

Figure 4.5: Matrix Multiplication

1. Trace ab with two entries (to a and to b), and two exits (to j and to h)
2. Trace jk with a single entry (to j), and two exits (to z and to b), and
3. Trace $hcfefg$ two entries (to h and to e) and three exits (to h, to e and to i). In this trace the inner most loop was unrolled, so each trace represents two iterations of the inner most loop. This is reflected in Figure 4.5c by adding an extra node f. Note that the transition from g to h is represented by an exit and an entry, not as an internal transition within the trace. This difference enables running multiple jobs concurrently executing the same trace code.

Thus, we divide the code, optimistically, into independent parts called traces that represent subsets of the execution graph. Each trace does not overlap with other traces in accessed variables, and represents a long sequence of instructions, including branch statements, that commonly execute in this pattern. Since a branch instruction has taken and not taken paths, the trace may contain one or both of the two paths according to the frequency of using those paths. For example, in biased branches, one of the paths is often considered; so it is included in the trace, leaving the other path outside the trace. On the other hand, in unbiased branches, both paths may be included in the trace. Therefore, a trace has multiple exits, according to the program control flow during its execution. A trace also has multiple entries, since a jump or a branch instruction may target one of the basic blocks that constructs it. The builder module orchestrates the construction of traces and distributes them over parallel threads. However, this may potentially lead to an out-of-order execution of the code, which we address through STM concurrency control (see Section 4.2). I/O instructions are excluded from parallel traces, as changing their execution order affects the program semantics, and they are irrevocable (i.e., at transaction aborts).

4.4.3 Parallel Traces

Upon detection of candidate trace for parallelization, the program is reconstructed as a producer-consumer pattern. In this pattern, two daemons threads are active, producer and consumer, which share a common fixed-size queue of jobs. Recall that a *job* represents a call to the synthetic methods executing the trace code with a specific set of inputs. The producer generates jobs and adds them in the queue, while the consumer dequeues the jobs and executes them. HydraVM uses a *Collector* module and an *Executor* module to process the jobs: the *Collector* has access to the generated traces and uses them as jobs, while the *Executor* executes the jobs by assigning them to a pool of core threads.

Figure 4.6 shows the overall pattern of the generated program. Under this pattern, we utilize the available cores by executing jobs in parallel. However, doing so requires handling of the following issues:

- Threads may finish in out of original execution order.

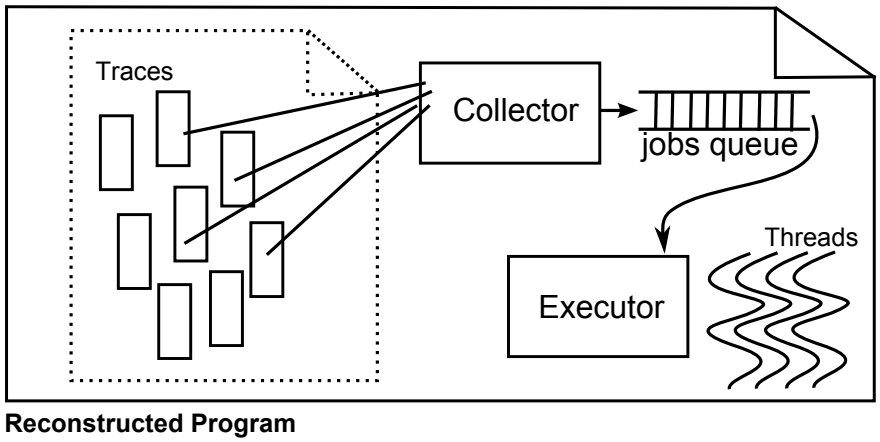


Figure 4.6: Program Reconstruction as a Producer-Consumer Pattern

- The execution flow may change at runtime causing some of the assigned traces to be skipped from the correct execution.
- Due to the differences between the actual execution flow in the profiling phase and the actual execution, memory access conflicts between concurrent accesses may occur. Also, memory arithmetic (e.g., arrays indexed with variables) may easily violate the program reconstruction (see example in Section 4.4.5).

To tackle these problems, we execute each job as a transaction. A transaction's changes are deferred until commit. At commit time, a transaction commits its changes if and only if: 1) it did not conflict with any other concurrent transaction, and 2) it is reachable under the execution.

4.4.4 Reconstruction Tuning

TM preserves data consistency, but it may cause degraded performance due to successive conflicts. To reduce this, the TM Manager provides feedback to the Builder component to reduce the number of conflicts. We store the commit rate, and the conflicting scenarios in the Knowledge Repository to be used later for further reconstruction. When the commit rate reaches a minimum preconfigured rate, the Builder is invoked. Conflicting traces are combined into a single trace. This requires changes to the control instructions (e.g., branching conditions) to maintain the original execution flow. The newly reconstructed version is recompiled and loaded as a *new class definition* at runtime.

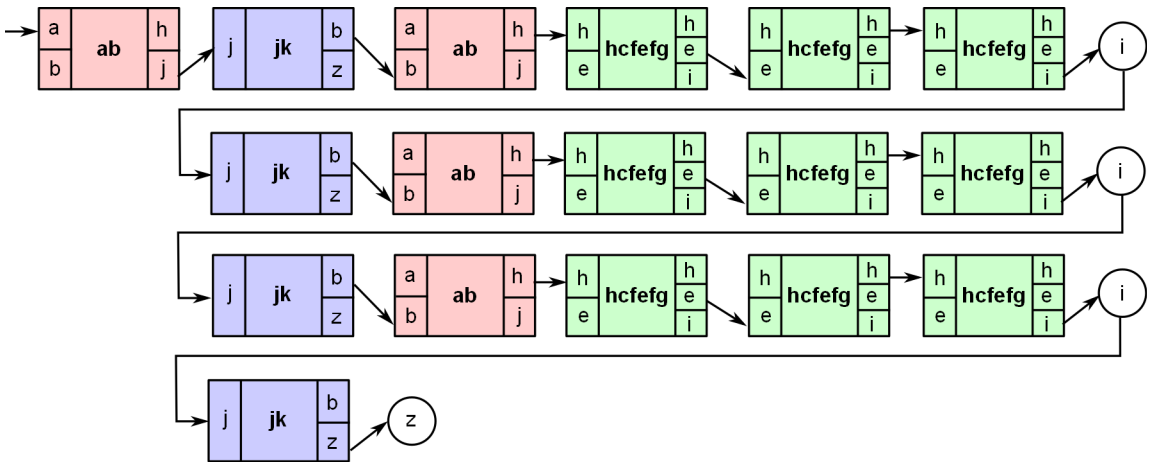


Figure 4.7: 3x3 Matrix Multiplication Execution using Traces Generated by profiling 2x2 Matrix Multiplication

4.4.5 Misprofiling

Profiling depends mainly on the program input. This input may not reflect some runtime aspects of the program flow (e.g., loops limits, biased branches). To illustrate this, we return to the matrix multiplication example in Figure 4.5a. Based on the profiling using 2x2 matrices, we construct the execution graph shown in Figure 4.5b. Now, recall our three traces ab, $hcfefg$, and jk, and assume we need to run this code for matrices 2x3 and 3x2. The Collector will assign jobs to the Executor, but upon the execution of the trace jk, the Executor will find that the code exits after j and needs to execute bs. Hence, it will request the Collector to schedule the job ab, with an entry to basic block b, in the incoming job set. Doing so allows us to extend the flow to cover more iterations. Note that the entry point must be sent to the synthetic method that represents the trace, as it should be able to start from any of its basic blocks (e.g., ab will start from b not a).

In Figure 4.7, traces are represented by blocks with their entries points on the left side, and exits on the right. The figure describes the execution using the traces extracted by profiling 2x2 matrix (See example at Section 4.4.1).

4.5 Implementation

4.5.1 Detecting Real Memory Dependencies

Recall that we use bytecode as the input, and concurrency refactoring is done entirely at the VM level. Compiler optimizations, such as register reductions and variable substitutions, increase the difficulty in detecting memory dependencies at the bytecode-level. For example,

```
y = 1                    y1 = 1
y += 2                   y2 = y1 + 2
x = y / 2                x1 = y2 / 2
```

Figure 4.8: Static Single Assignment form Example

two independent basic blocks in the source code may share the same set of local variables or loop counters in the bytecode. To overcome this problem, we transform the bytecode into the Static Single Assignment form (SSA) [24]. The SSA form guarantees that each local variable has a single static point of definition and is assigned exactly once, which significantly simplifies analysis. Figure 4.8 shows an example of the SSA form.

Using the SSA form, we inspect assignment statements, which reflect memory operations required by the basic block. At the end of each basic block, we generate a call to a *hydra_touch* operation that notifies the VM about the variables that were accessed in that basic block. In the second phase of profiling, we record the execution paths and the memory accessed by those paths. We then package each set of basic blocks in a trace. Traces should not be conflicting and access the same memory objects. However, it is possible to have such conflicts since our analysis uses information from past execution (which could be different from the current execution). We intentionally designed the data dependency algorithm to ignore some questionable data dependencies (e.g., loop index). This gives more opportunities for parallelization since if at run time a questionable dependency occurs, then TM will detect and handle it. Otherwise, such blocks will run in parallel and greater speedup is achieved.

4.5.2 Handing Irrevocable Code

Input and output instructions must be handled as a special case in the reconstruction and parallel execution as they cannot be rolled back. Traces with I/O instructions are therefore marked for special handling. The Collector never schedules such marked traces unless they are reachable – i.e., they cannot be run in parallel with their preceding traces. However, they can be run in parallel with their successor traces. This implicitly ensures that at most one I/O trace executes (i.e., only a single job of this trace runs at a time).

4.5.3 Method Inlining

Method inlining is the insertion of the complete body of a method in every place that it is called. In HydraVM, method calls appear as basic blocks, and in the execution graph, they appear as nodes. Thus, inlining occurs automatically as a side effect of the reconstruction process. This eliminates the time overhead of invoking a method.

Another interesting issue is handling recursive calls. The execution graph for recursion will appear as a repeated sequence of basic blocks (e.g., *abababab*...). Similar to method-

inlining, we merge multiple levels of recursion into a single trace, which reduces the overhead of managing parameters over the heap. Thus, a recursive call under HydraVM will be formed as nested transactions with lower depth than the original recursive code.

4.5.4 ByteSTM

ByteSTM [125] is STM that operates at the bytecode level, which yields the following benefits:

- Significant implementation flexibility in handling memory access at low-level (e.g., registers, thread stack) and for transparently manipulating bytecode instructions for transactional synchronization and recovery;
- Higher performance due to implementing all TM building blocks (e.g., versioning, conflict detection, contention management) at bytecode-level; and
- Easy integration with other modules of HydraVM (Section 4.4)

We modified the Jikes RVM to support TM by adding instructions, *xBegin* and *xCommit*, which are used to start and end a transaction, respectively. Each load and store inside a transaction is done transactionally: loads are recorded in a read signature and stores are sandboxed; stores are stored in a transaction-local storage, called the *write-set*. The address of any variable (accessible at the VM level) is added to the written signature. The read/write signature is represented using a Bloom filter [25] and used to detect read/write or write/write conflicts. This approach is more efficient than comparing transaction read-set and write-set of transactions, but it also increases false negatives. (With the correct signature size, the effect of false positives can be reduced – we do this.)

When a load is called inside a transaction, we first check the write-set to determine if this location has been written to before and if so, the value from the write-set is returned. Otherwise, the value is read from the memory and the address signature is added to the read signature. At commit time, the read signature and write the signature of concurrent transactions are compared, and if there is a conflict, the newer transaction is aborted and restarted again. If the validation shows no conflict, then the write-set is written to memory.

For a VM-level STM, greater optimizations are possible than that for non VM-level STMs (e.g., Deuce [101], DSTM2 [91]). At the VM level, data types do not matter; only their sizes do. This allows us to simplify the data structures used to handle transactions. One through eight-byte data types is handled in the same way. Similarly, all different data addressing is reduced to absolute addressing. Primitives, objects, array elements, and statics are handled differently inside the VM, but they are translated into an absolute address and a specific size in bytes. This simplifies and speeds-up the write-back process, since we only care about writing back some bytes at a specific address. This allows us to work at the field level and

at the array element level, which significantly reduces conflicts: if two transactions use the same object, but each use a different field inside the object, then no conflict occurs (similarly for arrays).

Another optimization is the ability to avoid the VM's Garbage Collector (GC). GC can reduce STM performance when attempting to free unused objects. Also, dynamically allocating new memory to be used by STM is costly. ByteSTM disables the GC for the memory used for the internal data structures that support STM, we statically allocate memory for STM, handle it without interruption from the GC, and manually recycle it. The new memory is allocated if there is a memory overflow. Note that if a hybrid TM is implemented in Java, then it must be implemented inside the VM. Otherwise, hybrid TM will violate invariants of internal data structures used inside the VM, leading to inconsistencies.

We also inline the STM code inside the load and store instructions and the newly added instructions $xBegin$ and $xCommit$. Thus, there is no overhead in calling the STM procedures in ByteSTM.

Each *job* has an order that represents its logical order in the sequential execution of the original program. To preserve the data consistency between jobs, STM must be modified to support this ordering. Thus, in ByteSTM, when a conflict is detected between two jobs, we abort the one with the higher order. Also, when a block with a higher order tries to commit, we force it to sleep until its order is reached. ByteSTM commits the block if no conflict is detected.

When attempting to commit, each transaction checks its order against the expected order. If they are the same, the transaction proceeds, validates its read-set, commit its write-set, and updates the expected order. Otherwise, it sleeps and waits for its turn. The validation is done by scanning the thread stack and registers, and collecting the accessed objects' addresses. Objects IDs are retrieved from the object copies and used to create a *transaction signature*, which represents the memory addresses accessed by the transaction. Transactional conflicts are detected using the intersection of transaction signatures. After committing, each thread checks if the next thread is waiting for its turn to commit, and if so, that thread is woken up. Thus, ByteSTM keeps track of the expected order and handles commit in a decentralized manner.

4.5.5 Parallelizing Nested Loops

Nested loops introduce a challenge for parallelization, as it is difficult to parallelize both inner and outer loops and imposes complexity to the system design. In HydraVM, we handle nested loops as nested transactions using the closed-nesting model [132]: aborting a parent transaction aborts all its inner transactions, but not vice versa, and changes made by inner transactions become visible to their parents when they commit, but those changes are hidden from outside world till the highest level parent's commit.

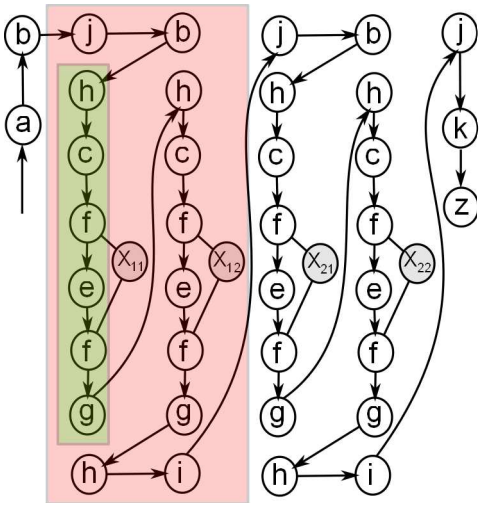


Figure 4.9: Nested Traces

1. Inner transactions share the read-set/write-set of their parent transactions;
2. Changes made by inner transactions become visible to their parent transactions when the inner transactions commit, but they are hidden from the outside world till the commit of the highest level parent;
3. Aborting an inner transaction aborts only its changes (not other sibling transactions, and not its parent transaction);
4. Aborting a parent transaction aborts all its inner transactions;
5. Inner transactions may conflict with each other and also with other, non-parent, higher-level transactions; and
6. Inner transactions are mutually ordered; i.e., their ages are relative to the first inner transaction of their parent. When an inner transaction conflicts with another inner transaction of a different parent, the ages of parent transactions are compared.

The use of closed nesting, instead other models such as linear nesting [132], is twofold:

1. Inner transactions run concurrently; conflict is resolved by aborting the higher age transaction.
2. We leverage the partial rollback of inner-transactions without affecting the parent or sibling transactions.

Although open nesting model [131] allows further increases in concurrency, open nesting contradicts our ordering model. In open nesting, inner transactions are permitted to commit

	Configurations
Processor	AMD Opteron Processor
CPU Cores	8
Clock Speed	800 MHz
L1	64 KB
L2	512 KB
L3	5 MB
Memory	12 GB
OS	Ubuntu 10.04, Linux

Table 4.1: Testbed and platform parameters for HydraVM experiments.

before its parent transaction, and if the parent transaction was aborted it uses a compensating action to revert the changes done by its committed inner transactions. However, in our model allowing inner transactions to commit will violate the ordering rule (lower transactions commit first), and preventing it from commit (i.e., wait until its chronological order) will cancel the idea behind open nesting.

Consider our earlier matrix multiplication example (See Section 4.4). From the execution string, $ab(jb(hcfefg)^2hi)^2jk$, we can create two nested traces: an outer trace $jb(hcfefg)^2hi$, and an inner trace $hcfefg$ (See Figure 4.9). The outer trace runs within a transaction, executing $jbhi$, that invokes a set of inner transactions $hcfefg$ after the execution of the basic block b.

4.6 Experimental Evaluation

Benchmarks. To evaluate HydraVM, we used five applications as benchmarks. These include a matrix multiplication application and four applications from the JOlden benchmark suite [34]: Minimum Spanning Tree (MST), tree add (TreeAdd), Traveling Salesman Problem (TSP), and Bitonic Sort (BiSort). The applications are written as sequential applications, though they exhibit data-level parallelism.

Testbed. We conducted our experiments on an 8-core multicore machine. Each core is an 800 MHz AMD Opteron Processor, with 64 KB L1 data cache, 512 KB L2 data cache, and 5 MB L3 data cache. The machine ran Ubuntu Linux.

Evaluation. Table 4.2 shows the result of the Profiler analysis on the benchmarks. The table shows the number of basic blocks, traces, and the average number of instructions per basic block. The lower part of the table shows the number of executed jobs by the Executor, and the maximum level of nesting during the experiments.

Using our techniques, we manage to split the sequential implementation of the benchmarks

Table 4.2: Profiler Analysis on Benchmarks

Benchmark	Matrix	TSP	BiSort	MST	TreeAdd
Avg. Instr. per BB.	4.29	4.2	4.75	3.7	4.1
Basic Blocks	31	77	24	52	10
Traces	3	12	5	3	4
Jobs	1001	1365	1023	12241	8195
Max Nesting	2	5	2	1	3

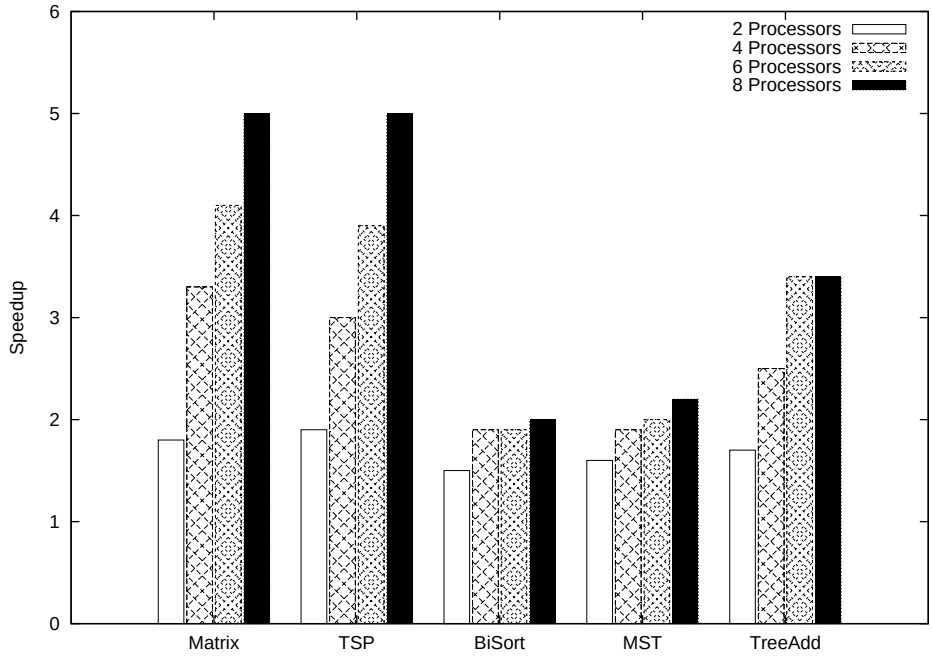


Figure 4.10: HydraVM Speedup

into parallel jobs that exploit the data-level parallelism. Figure 4.10 shows the speedup obtained for different number of processors. For matrix multiplication, HydraVM reconstructs the outer two loops into nested transactions, while the inner-most loop is formed as a trace because of the iteration dependencies. In TSP, BiSort, and TreeAdd, each multiple level of the recursive call is inlined into a single trace. For the MST benchmark, each iteration over the graph adds a new node to the MST, which creates inter-dependencies between iterations. However, updating the costs from the constructed MST and other nodes presents a good parallelization opportunity for HydraVM.

4.7 Discussion

We presented HydraVM, a JVM that automatically refactors concurrency in Java programs at the bytecode level. HydraVM extracts control-level parallelism by reconstructing program as independent traces. Loops, as a special case of traces, is included into the reconstruction procedure which supports data-level parallelism. Although, our goal targets extracting code traces, most of the applications spend most of time executing loops. Loops have interesting features such as: symmetric code blocks, data level parallelism, recurrences, and induction variables. In the next chapter, we focus on loops as a unit for parallelization.

Online profiling and adaptive compilation provide transparent execution of the programs. Nevertheless, it adds an overhead to the runtime. Such overhead is non-negligible for short life applications, or ones with unpredictable execution paths. Static analysis of the code could be beneficial for providing an initial guess of hot-spot regions of the code, and build dependency relations between code-blocks. Adding a pre-execution static analysis and employing data dependency analysis could enhance the code generation and reduce transactional overhead, we followed this approach in Lerna.

Chapter 5

Lerna

In this chapter, we present Lerna, a system that automatically and transparently detects and extracts parallelism from sequential code. Lerna is cross-platform and independent of the programming language, and does not require the analysis of the application's source code, it simply takes its intermediate representation compiled using *LLVM* [108], the well-known compiler infrastructure, as input and produces ready-to-run parallel code as output[1], thus finding its best fit with (legacy) sequential applications. This approach makes Lerna independent also of the specific hardware used.

Similar to HydraVM, the parallel execution exploits memory transactions to manage concurrent and out-of-order memory accesses. While, HydraVM presents an initial concept of a virtual machine that exploits in-memory transactions to parallelize traces, Lerna focus on parallelizing loops, which usually contains the hotspot sections in many applications. Recall that HydraVM reconstructs the code at runtime through recompilation and reloading class definition, and it is obligated to run the application through the virtual machine. Unlike HydraVM, Lerna does not change the code at runtime through recompilation, which enable us to compile the code to its native representation. Nevertheless, Lerna supports adaptive execution of transformed programs through changing some key perfomance parameters of its runtime library (See Section 5.8). Finally, the extensive profiling phase at HydraVM relies on establishing a relation between basic blocks and their accessed memory addresses, which limits its usage to small size applications. In Lerna, we replaced that with a static alias analysis phase combined with light offline profiling step that complement our static analysis.

Lerna is a complete system, which overcome all the above limitations and embeds system and algorithmic innovations to provide high performance. Our experimental study involves the transformation of 10 sequential applications into parallel. Results showed an average of 2.7× speedup for micro-benchmarks and 2.5× for the macro-benchmarks.

[1] Lerna supports the generation of native executable as output.

5.1 Challenges

Despite the high-level goal showed above, without fine-grain optimizations and innovations, deploying TM-style transactions to blocks of code that are prone to be parallelized leads the application performance to be slower (often much slower) than the sequential, non-instrumented execution. As an example of that, a blind parallelization of a loop (Figure 5.1a) would mean wrapping the whole body of the loop within a transaction (Figure 5.1b). By doing so, we let all transactions conflict with each other on, at least, the increment of the variable c or i[2]. In addition: variables that have been never modified within the loop may be transactionally accessed; the transaction commit order should be the same as the completion order of the iterations if they would have executed sequentially; aborts could be costly, as it involves retrying the whole transaction including local processing work. The combination of these factors nullifies any possible gain due to parallelization, thus letting the application pay just the overhead of the transactional instrumentation and, as a consequence, providing performance slower than sequential execution.

Lerna does not suffer from the above issues (as showed in Figure 5.1c). It instruments a small subset of code instructions, which is enough to preserve correctness, and optimizes the processing by a mix of static optimizations and dynamic tuning. The first include: loop simplification, induction variable reduction, removal of non-transactional work from the context to restore after a conflict, exploitation of the symmetry of executed parallel code, and an optimized in-order commit of transactions. Regarding the latter, Lerna provides an adaptive runtime layer to improve the performance of the parallel execution. This layer is fine-tuned by collecting feedbacks from the actual execution in order to capture the best settings of the key performance parameters that most influence the effectiveness of the parallelization (e.g., number of worker threads, size and number of parallel jobs).

The results are impressive. We evaluated Lerna's performance using a set of 10 applications including micro-benchmarks from the RSTM [2] framework, STAMP [37], a suite of sequential applications designed for evaluating in-memory concurrency controls, and Fluidanimate [133], an application performing physics simulations. The reason we selected them is because they provide (except for Fluidanimate) also a performance upper-bound for Lerna. In fact, they are released with a version that provides synchronization by using manually defined, and optimized, transactions. This way, besides the speedup over the sequential implementation, we can show the performance of Lerna against the same application with an efficient, hand-crafted solution. Lerna is on average 2.7× faster than the sequential version using micro-benchmarks (with a pick of 3.9×), and 2.5× faster considering macro-benchmarks (with a top speedup of one order of magnitude reached with STAMP).

Lerna is self-contained, completely automated and transparent system that makes sequential applications parallel. Thanks to an efficient use of transactional blocks, it finds its sweet

[2]Libraries that require programmer interaction, as OpenMP, already offer programming primitives to handle loops increments.

```
c = min;
while(i < max){
        i++;
        c = c + 5;
        ... local processing work ...
        if(i < j)
                k = k + c;
}
```

(a) Loop with data dependency

```
c = min;
while(i < max){
    atomic{
       TX_WRITE(i, TX_READ(i) + 1);
       TX_WRITE(c, TX_READ(c) + 5);
       ... local processing work ...
       if(TX_READ(i) < TX_READ(j))
           TX_WRITE(k,
             TX_READ(k) + TX_READ(c));
    }
}
```

(b) Loop with atomic body

```
c = min;
while(i < max){
    i++;
    parallel(i){
        c = min + i*5;
        ... local processing work ...
        atomic{
        if(i < j)
            TX_INCREMENT(k, c);
        }
    }
}
```

(c) Loop with parallelized body

Figure 5.1: Lerna's Loop Transformation: from Sequential to Parallel.

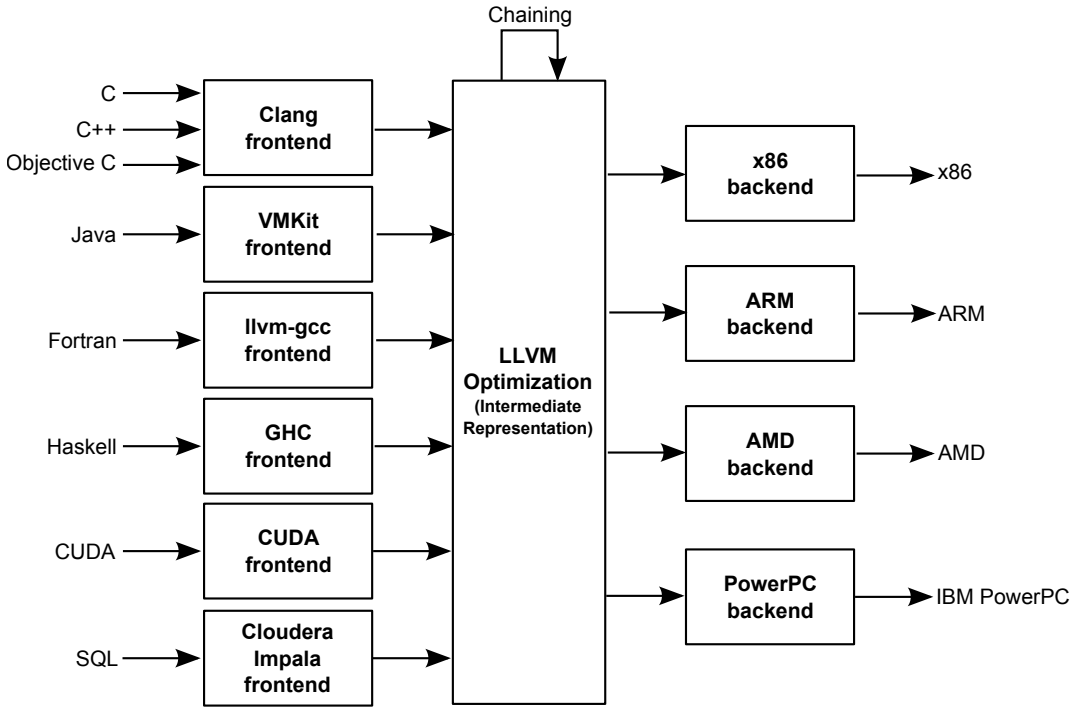


Figure 5.2: LLVM Three Layers Design

spot with applications involving data sharing but not simply those with partitioned memory access.

5.2 Low-Level Virtual Machine

Low-Level Virtual Machine (LLVM) is a modular and reusable collection of libraries, with well-defined interfaces, that define a complete compiler infrastructure. It represents a middle layer between front-end language-specific compilers, and back-end instruction sets generators. LLVM offers an Intermediate Representation (IR), bytecode, that can be optimized and transformed into more *efficient* IR. Optimizers can be chained to provide multi-level of optimizations at different levels of scopes (modules, call graph, function, loop, region, and basic block). LLVM supports a language-independent instruction set and data types in the form of Single Static Assignment (SSA).

Such separation of layers help developers to write front-ends to get use of underlying compiler optimizations. At the current stage, LLVM supports most of the widely used languages such as: C, C++, Objective-C, Java, Fortran, Haskell, and Ruby – the full list at [3]. The most common LLVM front-end is Clang which support compiling C, Objective-C and C++ codes. CLange is supported by Apple as a replacement for C/Objective-C compiler in the GCC system. Likewise, several platforms including: x86/x86-64, AMD, ARM, SPARC, MIPS,

and Nvidia, have LLVM back-end generators for its instruction sets – See Figure 5.2. These factors together builds a large community for LLVM as a popular compiler infrastructure. With LLVM, researchers build their optimizations on the intermediate layer using LLVM byte code, and have their optimizations available for large set of languages/platforms.

5.3 General Architecture and Workflow

Lerna is deployed as a container of:

- an *automated software tool* that performs a set of transformations and analysis steps (also called *passes* in accordance with the terminology used by LLVM) that run on the LLVM intermediate representation of the original binary application code. It produces a refactored multi-threaded version of the input program that can run efficiently on multiprocessor architectures;
- a *runtime library* that is linked dynamically to the generated program, and is responsible for: *1)* organizing the transactional execution of dispatched jobs so that the original program order (i.e., the chronological order) is preserved; *2)* selecting (adaptively) the most effective number of worker threads according to the actual setup of the runtime environment, and based on the feedbacks collected from the online execution; *3)* scheduling jobs to threads according to threads' characteristics (e.g., stack size, priority); and *4)* performing memory housekeeping and releasing computational resources.

Figure 5.3 shows the architecture and the workflow of Lerna. Lerna operates at the LLVM intermediate representation of the input program, thus it does not require the application to be written in any specific programming language. However, Lerna's design does not preclude the programmer from providing hints that can be leveraged to make the refactoring process more effective. In this work we provide the fully automated process without considering any programmer intervention, and we discuss how to exploit the prior application knowledge in Section 5.8. Lerna's workflow includes the following three steps in this order: *Code Profiling*, *Static Analysis*, and *Runtime*.

In the first step, our software tool executes the original (sequential) application by activating our own profiler that collects some important parameters (e.g., execution frequencies) later used by the Static Analysis.

The goal of the Static Analysis is to produce a multi-threaded (also called reconstructed) version of the input program. This process evolves by following the below passes:

- *Dictionary Pass.* It scans the input program to provide a list of the *accessible* (i.e., which is not either a system-call or a native-library call) functions of the bytecode (or

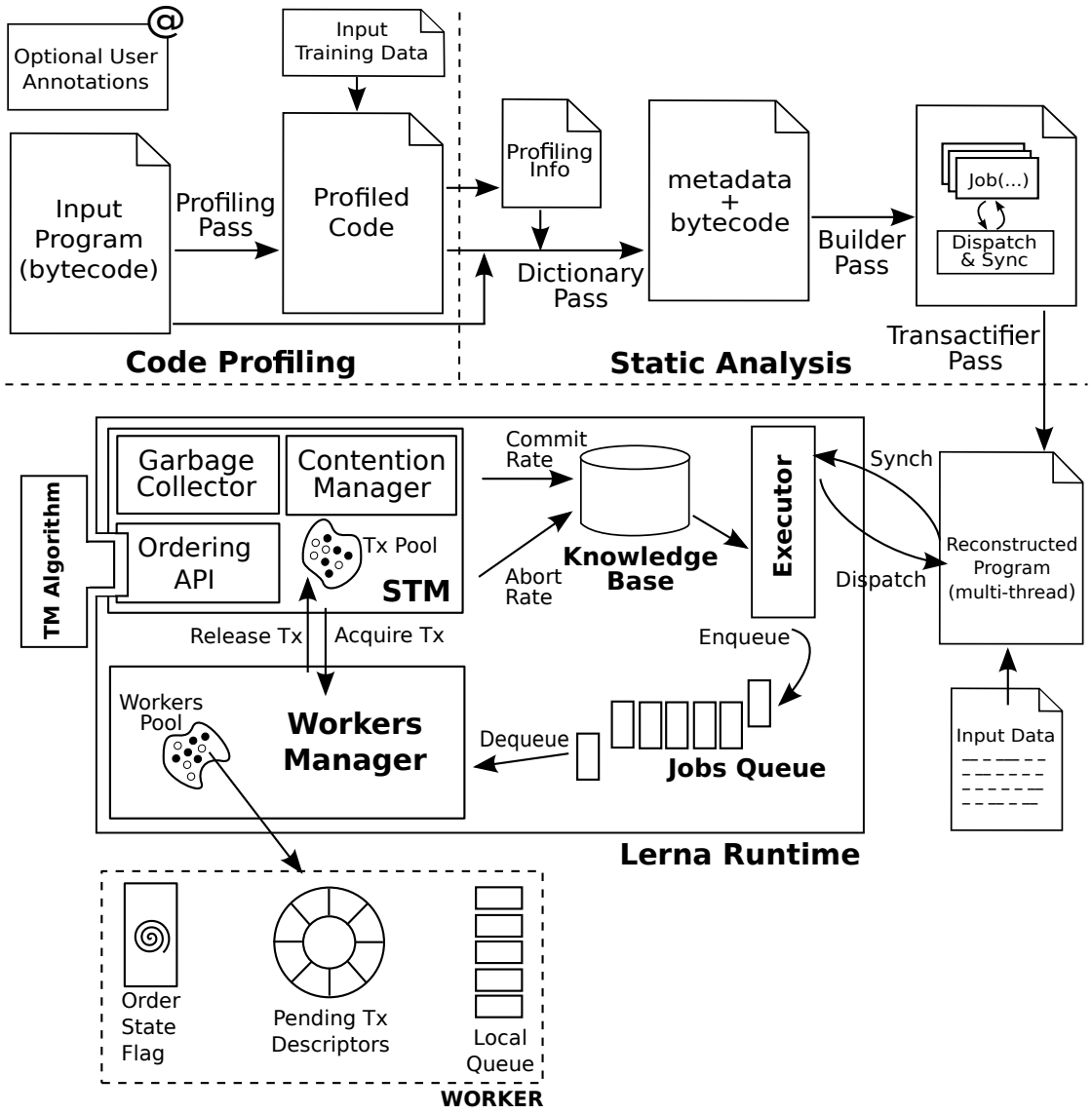


Figure 5.3: Lerna's Architecture and Workflow

the *bitcode* as named by LLVM) that we can analyze to determine how to transform. By default, any call to an external function is flagged as *unsafe*. This information is important because transactions cannot contain unsafe calls as they may include irrevocable (i.e., which cannot be further aborted) operations, such as I/O system calls.

- *Builder Pass*. It detects the code eligible for parallelization; it transforms this code into a callable synthetic method; and it defines the transaction's boundaries (i.e., where the transaction begins and ends).

- *Transactifier Pass*. It applies the alias analysis [47] (i.e., it detects if multiple references point to the same memory location) and some memory dependency techniques (e.g., given a memory operation, extracts the preceding memory operations that depend on it) to reduce the number of transactional reads and writes. It also provides the instrumentation of memory operations invoked within the body of a transaction by wrapping them into transactional calls for read, write or allocate.

Once the Static Analysis is complete, the reconstructed version of the program is linked to the application through the Lerna runtime library, which is mainly composed of the following three components:

- *Executor*. It dispatches the parallel jobs and provides the exit of the last job to the program. To exploit parallelism, the executor dispatches multiple jobs at-a-time by grouping them as a batch. Once a batch is complete, the executor simply waits for the result of this batch. Not all the jobs are enclosed in a single batch, thus the executor could need to dispatch more jobs after the completion of the previous batch. If no more job should be dispatched, the executor finalizes the execution of the parallel section.

- *Workers Manager*. It extracts jobs from a batch and it delivers ready-to-run transactions at available worker threads.

- *TM*. It provides the handlers for transactional accesses (read and write) performed by executing jobs. In case a conflict is detected, it also behaves as a contention manager by aborting the conflicting transactions with the higher chronological order (this way the original program's order is respected). Also, it handles the garbage collection of the memory allocated by a transaction, after it completes.

The runtime library makes use of two additional components: the *jobs queue*, which stores the (batch of) dispatched jobs until they are executed; and the *knowledge base*, which maintains the feedbacks collected from the execution in order to enable the adaptive behavior.

5.4 Code Profiling

Lerna uses the code profiling technique for identifying hotspot sections of the original code, namely those most visited during the execution. This information is fundamental for letting the refactoring process focus on the real parts of the code that are fruitful to parallelize (e.g., it would not be effective to parallelize a for-loop with only two iterations).

To do that, we consider the program as a set of *basic blocks*, where each basic block is a sequence of non-branching instructions that ends either with a branch instruction (conditional or non-conditional) or a return. Given that, any program can be represented as a graph in which nodes are basic blocks and edges reproduce the program control flow (an example of such a graph is shown in Figure 5.5). Basic blocks are easily determined from the bytecode (see Figure 5.4).

In this phase, our goal is to identify the context, frequency and reachability of each basic block. To determine that information, we profile the input program by instrumenting its bytecode at the boundaries of any basic blocks to detect whenever a basic block is reached. This code modification does not affect the behavior of the original program. We call this version of the modified program *profiled bytecode*.

5.5 Program Reconstruction

In the following, we illustrate in detail the transformation from sequential code to parallel made during the static analysis phase. The LLVM intermediate representation (i.e., the bytecode) is in the static single assignment (SSA) form. With SSA, each variable is defined before it is used, and it is assigned exactly once. Figure 5.4 shows LLVM intermediate representation of the loop Figure 5.1a. The code at Figure 5.4 is in SSA form, and is divided into sets of *basic blocks*. Each basic block starts with an optional label; it is composed of LLVM assembly instructions; and it ends with a branching instruction (*terminator*).

5.5.1 Dictionary Pass

In the dictionary pass, a full bytecode scan is performed to determine the list of accessible code (i.e., the dictionary) and, as a consequence, the external calls. Any call to an external function that is not included in the input program prevents the enclosing basic block from being included in the parallel code. However, the user can override this rule by providing a list of *safe* external calls. An external call is defined as *safe* if:

- It is revocable (e.g., it does not perform input/output operations);
- It does not affect the state of the program; and

```
entry:
  %retval = alloca i32, align 4
  br label %while.cond
while.cond:
; preds = %if.end, %entry
  %0 = load i32* %i, align 4
  %1 = load i32* %max, align 4
  %cmp = icmp slt i32 %0, %1
  br i1 %cmp, label %while.body, label %while.end
while.body:                                       ; preds = %while.cond
  %2 = load i32* %i, align 4
  %inc = add nsw i32 %2, 1
  store i32 %inc, i32* %i, align 4
  call void @_Z19do_local_processingv()
  %3 = load i32* %i, align 4
  %4 = load i32* %j, align 4
  %cmp1 = icmp slt i32 %3, %4
  br i1 %cmp1, label %if.then, label %if.end
if.then:                                          ; preds = %while.body
  %5 = load i32* %k, align 4
  %add = add nsw i32 %5, 1
  store i32 %add, i32* %k, align 4
  br label %if.end
if.end:
; preds = %if.then, %while.body
  br label %while.cond
while.end:                                        ; preds = %while.cond
  %6 = load i32* %retval
  ret i32 %6
```

Figure 5.4: The LLVM Intermediate Representation using SSA form of Figure 5.1a.

- It is thread safe.

A common example of safe calls are stateless random generators, or mathematical basic functions such as trigonometric functions.

5.5.2 Builder Pass

This pass is one of the core steps made by the refactoring process because it takes the code to transform (as output of the profiling phase) and makes it parallel by matching the outcome of the dictionary pass. In fact, if the profiler highlights an often invoked basic block that contains calls not in the dictionary, then the parallelization cannot be performed on that basic block.

In this thesis we focus on loops as the most appropriate blocks of code for being parallelized. However, our design is applicable (unless stated otherwise) for any independent sets of basic blocks. The actual operation of building the parallel code takes place after the following two transformations.

- *Loop Simplification analysis.* A *natural loop* has one entry block *header* and one or more back edges (*latches*) leading to the header. The predecessor blocks for the loop header are called *pre-header* blocks. We say that a basic block α dominates another basic block β if every path in the code β go through α. The *body* of the loop is the set of basic blocks that are dominated by its header, and reachable from its latches. The *exits* are basic blocks that jump to a basic block that is not included in the loop body. In Figure 5.4, the *entry* block is the loop pre-header, while its header is *while.cond*. The loop has one latch (i.e., *if.end*), and a single exit *while.end* (from *while.cond*). The loop body is the set of blocks *while.body*, *if.then* and *if.end*. A *simple loop* is a natural loop, with a single pre-header and single latch; and its index (if exists) starts from zero and increments by one.

  We apply the loop simplification to put the loop into its simplest form. Examples of natural and simple loops are reported in Figures 5.5 (a) and (b), respectively. In Figure 5.5 (a), the loop header has two types of predecessors, external basic blocks from outside of the loop, and one of the body latches. Putting this loop in its simple form requires adding: *i)* a single pre-header and changing the external predecessors to jump to the pre-header; and *ii)* an intermediate basic block to isolate the second latch from the header.

- *Induction Variable analysis.* An *induction variable* is a variable within a loop whose value changes by a fixed amount every iteration (i.e., the loop index) or is a linear function of another induction variable. Affine (linear) memory accesses are commonly used in loops (e.g., array accesses, recurrences). The index of the loop, if one exists, is often an induction variable, and the loop can contain more than one induction variable. The

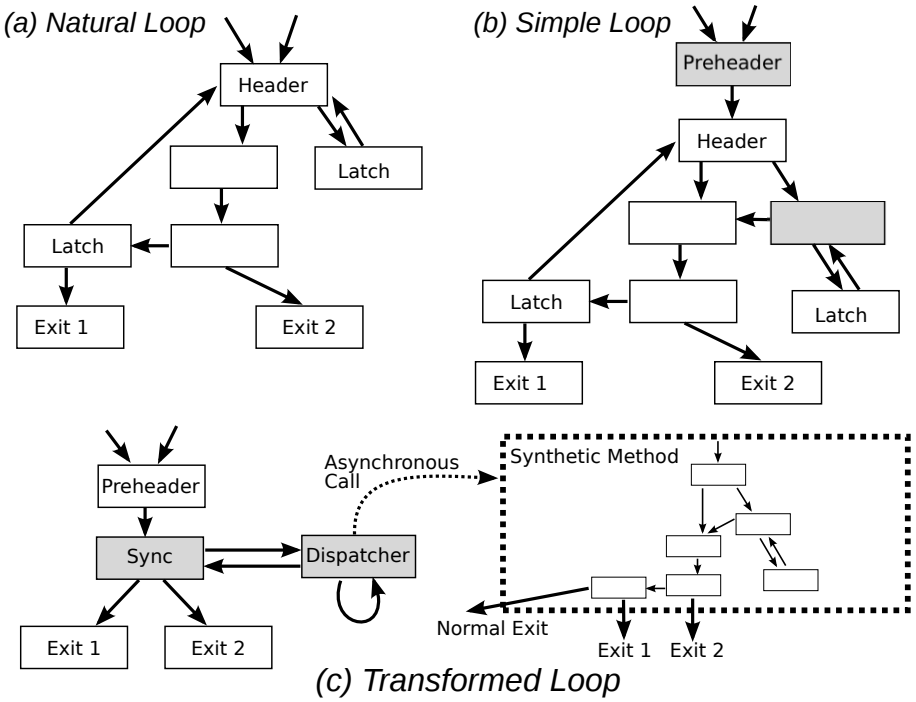


Figure 5.5: Natural, Simple and Transformed Loop

> *induction variable substitution* is a transformation to rewrite any induction variable in the loop as a closed form (function) of its index. It starts by detecting the candidate induction variables, then it sorts them topologically and creates a closed symbolic form for each of them. Finally, it substitutes their occurrences with the corresponding symbolic form.

As a part of our transformation, a loop is simplified, and its induction variable (i.e., the index) is transformed into its canonical form where it starts from zero and is incremented by one. A simple loop with multiple induction variables is a very good candidate for parallelization. However, any induction variables introduce dependencies between iterations, which are not desirable to maximize parallelism. To solve this problem, the value of such induction variables is calculated as a function of the index loop prior to executing the loop body, and it is sent to the synthetic method as a runtime parameter. This approach avoids unnecessary conflicts on the induction variables.

Next, we extract the body of the loop as a synthetic method. The return value of the method is a numeric value representing the exit that should be used. The addresses of all variables accessed within the loop body are passed as parameters to the method.

The loop body is replaced by two basic blocks: *Dispatcher* and *Sync*. In the *Dispatcher*, we prepare the arguments for the synthetic method, calculate the value of the loop index and invoke an API of our library, named *lerna_dispatch*, providing it with the address of the

Tx1	Tx2
Y[1] := 0	Y[2] := 0
Z[1] := X[1]	Z[2] := X[2]

(a) Symmetric Transactions

Tx1	Tx2
Y[1] := 0	
	X[1] := Y[1]
Z[1] := X[1]	

(b) Normal Transactions

Figure 5.6: Symmetric vs Normal Transactions

synthetic method and the list of the just-computed arguments. Each call to *lerna_dispatch* adds a job to our internal jobs queue, but it does not start the actual execution of the job. The *Dispatcher* keeps dispatching jobs until our API decides to stop. When it happens, the control passes to the *Sync* block. *Sync* immediately blocks the main thread and waits for the completion of the current jobs. Figure 5.5 (c) shows the control flow diagram (CFG) for the loop before and after transformation.

Regarding the exit of a job, we define two types of exits: *normal exit* and *breaks*. A normal exit occurs when a job reaches the loop latch at the end of its execution. In this case, the execution should go to the header and the next job should be dispatched. If there are no more dispatched jobs to execute and the last one returned a normal exit, then the *Dispatcher* will invoke more jobs. On the other hand, when the job exit is a break, then the execution needs to leave the loop body, and hence ignore all later jobs. For example, assume a loop with N iterations. If the Dispatcher invokes B jobs before moving to the Sync, then $\lceil N/B \rceil$ is the maximum number of transitions that can happen between Dispatcher and Sync.

Summarizing, the Builder Pass turns the execution model into the job-driven model, which can exploit parallelism. This strategy abstracts the processing from the way the code is written.

5.5.3 Transactifier Pass

After turning the bytecode into executable jobs, we employ additional passes to encapsulate jobs into transactions. Each synthetic method is demarcated by *tx_begin* and *tx_end*, and any memory operation (i.e., load, stores or allocation) within the synthetic method is replaced by the corresponding transactional handler.

It is quite common that memory reads are numerous (and outnumber writes), thus it would be highly beneficial to minimize those performed transactionally. That is because, the read-set maintenance and the validation performed at commit time for preserving the correctness of the transaction, which iterates over the read-set, is the primary source of TM's overhead. Several hardware prototypes have been proposed to enhance TM read-set management [38, 166]. Alternatively, in our work the transactifier pass eliminates unnecessary transactional reads, thus significantly improving the performance of the transaction execution due to the

following reasons:

- direct memory read is even three times faster than transactional read [38, 166]. In fact, reading an address transactionally requires: *1)* checking if the address has already been written before (i.e., check the write-set); *2)* adding the address to the read-set; and *3)* returning the address value to the caller.
- the size of the read-set is limited, thus extending it requires copying entries into a larger read-set, which is costly. Keeping the read-set small reduces the number of resize operations.
- read-set validation is mandatory during the commit. The smaller the read-set, the faster the commit operation.

In our model, concurrent transactions can be described as "symmetric", which means that the code executed in all active transactions is the same. That is because each transaction executes one or more iterations of the same loop. Figure 5.1 shows an example of symmetric transactions. We take advantage of this characteristic by reducing the number of transactional calls as follows.

Clearly, local addresses defined within the scope of the loop are not required to be accessed transactionally. On the other hand, global addresses allow iterations to share information, and thus they need to be accessed transactionally. We perform the *global alias analysis* as a part of our transactifier pass to exclude some of the loads to shared addresses from the instrumentation process.

To reduce the number of transactional reads, we apply the global alias analysis between all loads and stores in the transaction body. A load operation that will never alias with any store operation does not need to be read transactionally. For example, when a memory address is always loaded and never written in *any path* of the symmetric transaction code, Figure 5.6a, then the load does not need to be performed transactionally. In Figure 5.1, concurrent transactions execute symmetric iterations of the loop. Although j is read within the transactions, we do not have to read it transactionally as it can never be changed by any of the transactions produced from the same loop (thus the only allowed to run concurrently). Note that this technique is specific for parallelizing loops and cannot be applied to the normal transaction processing where all transactions do not necessarily execute the same code as for the symmetric transactions.

In contrast, in Figure 5.6b we show an example of two concurrent non-symmetric transactions, thus executing different transaction bodies. Also in this case each transaction has a load operation that does not alias with other stores but when the two transactions (with different code paths) run concurrently they can produce wrong result. This is because these transactions are not symmetric and thus the read must be done transactionally.

Transactions may contain calls to other functions. As these functions may manipulate memory locations, they must be handled. Whenever possible, we try to inline the called functions; otherwise we create a transactional version of the function called within a transaction. In the latter case, instead of calling the original function, we call its transactional version. Inlined functions are preferable because they permit the detection of dependencies between variables, which can be leveraged to reduce transactional calls, or the detection of dependent loop iterations, which is useful to exclude them from the parallelization.

Finally, to avoid unnecessary overhead in the presence of single-threaded computation or a single job executed at a time, we create another non-transactional version of the synthetic method. This way we provide a fast version of the code without unnecessary transactional accesses.

5.6 Transactional Execution

Transactional memory algorithms differ in the memory versioning techniques, i.e., undo-log or write-buffer, and in the conflict detection, i.e., lazy or eager. In write-buffer algorithms, transactional loads are recorded in a read-set and stores are and-boxed, which means that each store is not written to the original address but it is kept into a local storage called the write-set until commit. When a load is called inside a transaction, the write-set is first checked to determine if this location has been written before and, if so, the value from the write-set is returned. Otherwise, the value is read from the memory and the address is added to the read-set. At commit time, the read-set is validated to make sure that it is still consistent. If the validation shows no conflict, then the write-set is written back to the shared memory and the changes become visible to all. On the other hand, undo-log algorithms expose memory changes to the main memory right after the transactional write, and they keep the old values in a local log to be restored upon transaction abort.

Contention between transactions occur when two transactions access the same address and one of them is a writer. Eager contention detection is done by associating a lock with each memory address (a lock can cover multiple addresses). A transaction acquires locks on its written addresses at encounter time. Conflicts are detected when a transaction tries to access a locked address. Alternatively, in lazy contention detection, each address has a version record. Transaction stores the version of its read addresses. Succeeded transactions modify the version of their written addresses at commit time. Transaction is aborted when it finds a different version number than the one recorded. Validation of memory addresses accessed transactionally is done either by performing addresses comparison [59], or by checking if the values have changed [53].

Table 5.1 shows different design choices of some known transactional memory implementations.

Our design is decoupled from the specific TM implementation used but it requires in-order

		Version	
		Eager	Lazy
Contention	Eager	TinySTM [69], LogTM [129] UTM [10], McRT-STM [165]	LTM [10], TinySTM [69], SwissTM [63]
	Lazy		TL2 [59], TCC [81], NOrec [53]

Table 5.1: Transactional Memory Design Choices

commit. To allow the integration of further TMs, we identified the following requirements needed to support ordering:

Supervised Commit. Threads are not allowed to commit once they complete their execution. Instead, there must be a single stakeholder at-a-time that accomplished transaction commits, namely the *committer*. It is not necessary having a dedicated committer because worker threads can take over this role according to their age. For example, the thread executing the transaction with the lowest age could be the committer and thus it is allowed to commit. While a thread is committing, other threads can proceed by executing next transactions speculatively, or wait until the commit completes. Allowing threads to proceed with their execution is risky because it can increase the contention probability given that the life of an uncommitted transaction enlarges (e.g., the holding time of their locks increases, or the timestamp validity decreases), therefore this speculation must be limited by a certain (tunable) threshold. Upon a successful commit, the committer role is delegated to the subsequent thread with lowest age (which may be the same thread in some cases). This strategy allows only one thread to commit its transaction(s) at a time.

An alternative approach is to use a single committer (as in [120]) to monitor the completed transactions, and to permit non-conflicting threads to commit in parallel by inspecting their read- and write-set. Although this strategy allows for concurrent commits, the performance is bounded by the committer execution time.

Age-based Contention Management (CM). Algorithms with eager conflict detection (i.e., at encounter time) should favor transactions with lower age (i.e., that encapsulate older iterations), while algorithms that use lazy conflict detection (i.e., at commit time) should employ an aggressive CM that favors the transaction that is committing using the single committer. Note that, for value-based validation with eager versioning, it is possible for earlier transactions to wrongly commit after it read from a speculative iteration. To solve this, during the commit phase, the read-set must be compared with the write-sets of completed transactions.

Memory Versioning. For eager versioning, if the implementation uses eager CM, then it prevents speculative iterations from affecting older iteration because they will collide. On the other hand, Lazy CM may cause earlier iterations to read from speculative iterations. However, thanks to the single committer, the former iteration will detect the conflict and both transactions will be aborted. Lazy versioning implementations hide their changes from

```
for(int i=0;i<100;i++){
    ...
    if(some_condition)
        counter++;
    ...
}
```

(a) Conditional increments

```
while(proceed) {
    ...
    counter++;
    ...
}
```

(b) No induction variable

Figure 5.7: Conditional Counters

other transactions, thus no modification is required for this category.

5.6.1 High-priority Transactions

A transaction performs a read-set validation at commit time to preserve correctness. That is needed to ensure that its read-set has not been overwritten by any other committed transaction. Let Tx_n be a transaction that has just started its execution, and let Tx_{n-1} be its immediate predecessor (i.e., Tx_{n-1} and Tx_n process consecutive iterations of a loop). If Tx_{n-1} has been committed before that Tx_n performs its first transactional read, then we can avoid the read-set validation of Tx_n when it commits. That is because Tx_n is now the highest priority transaction at this time, so no other transaction can commit its changes to the memory. We do that by flagging Tx_n as an *irrevocable transaction*. Similarly, a transaction is flagged as *high-priority* if: *i)* it is the first, thus it does not have a predecessor; *ii)* it is a retried transaction of the single committer thread; *iii)* there is a sequence of transactions with consecutive age running on the same thread.

This optimization reduces the commit time by just writing the write-set values to the memory.

5.6.2 Transactional Increment

Figure 5.7 illustrates a common situation, which is the *counter*. Loops with counters hamper parallelism because they create data dependencies between iterations, even non-consecutive iterations, which produces a large amount of conflicts. The induction variable substitution cannot produce a closed form (function) of the loop index (if it exists). If a variable is incremented (or decremented) based on any arbitrary condition and its value is used only after the loop completes the whole execution, then it is eligible for the *Transactional Increment* optimization.

In addition to the classical transactional read and write (*tx_read* and *tx_write*), we propose

a new transactional primitive, the *transactional increment*, to enable the parallelization of loops with irreducible counters. This type of counter can be detected during our transformations. Within the transactional code, a store S_t is eligible for our optimization if it aliases only with one load L_d, and it writes a value that is based on the return value of L_d. The load, change, and store operations are replaced with a call to *tx_increment*, which receives the address and the value to increment. We propose two ways to implement *tx_increment*:

- Using an atomic increment to the variable, and storing the address to the transaction's meta-data. The atomic operation preserves data consistency; however, it affects the shared memory before the transaction commits. To address this issue, aborted transactions compensate all accessed counters by performing the same increment but with the inverse value.
- By storing the increments into thread-specific meta-data. At the end of each *Sync* operation, threads coordinate with each other to expose the aggregated per-thread increments of the counter. This method is appropriate for floating point variables, which cannot be updated atomically on commodity hardware.

Using this approach, transactions will not conflict on this address, and the correct value of the counter will be in memory after the completion of the loop (See Figure 5.1c).

5.7 Algorithms

Lerna currently integrates four TM implementations with different designs: NOrec [53], which executed commit phases serially without requiring any ownership record; TL2 [59], which allows parallel commit phases but at the cost of maintaining an external data structure for storing meta-data associated with the transactional objects; *UndoLog* [4] with visible readers, which uses encounter time versioning and locking for accessed objects and maintains a list of accessors transactions; and STMLite [120], which replaces the need for locking objects and maintaining a read-set with the use of signatures. STMLite is the only TM library designed for supporting loop parallelization.

5.7.1 Ordered NOrec

In the current implementation we used NOrec [53] as a TM library. It is an algorithm that offers low memory access overhead with constant amount of global meta-data. Unlike most STM algorithms, NOrec does not associate ownership records (e.g., locks or version number) with accessed addresses; instead, it employs a value-based validation technique during commit. A characteristic of this algorithm is that it permits a single committing writer

at a time, which is in general not desirable but it matches the need of Lerna's concurrency control: having a single committer (See Section 5.6). For this reason we decided to rely on NOrec as the default STM implementation for Lerna because of its small memory footprint and because it matches our ordering conditions (we need a single committer as limitation at NOrec, it is one of the requirement for ordering transactions). Our modified version of NOrec manages which transaction should be the single committer according to the chronological order (i.e., age).

5.7.2 Ordered TinySTM

TinySTM algorithm uses encounter time locking (ETL) and comes with two memory access strategies: write-through and write-back. TinySTM uses timestamps for transactions to ensure a consistent view of memory; we exploit this timestamp to represent the *age* of transactions. Using an aggressive age-based contention manager, transactions can be ordered according to the chronological order of their executing code. With TinySTM, transactions conflict at access time; this allows early detection of conflicting iterations. The choice of write-through or write-back strategy is workload specific; at low-contention write-through provides a fast path execution, while write-back is more suitable for high-contention as it exhibits lower overhead abort procedure.

5.7.3 Other Algorithms

For completeness, we included ordered version of TL2 and UndoLog algorithms. Both algorithms follow two different design choices: write-back and undo-log, respectively. However, the integration of more TM algorithm using the techniques explained in Section 5.6 is straightforward.

5.8 Adaptive Runtime

The Adaptive Optimization System (AOS) [12] is a general virtual machine architecture that allows online feedback-directed optimizations. In Lerna, we apply the AOS to optimize the runtime environment by tuning some important parameters (e.g., the batch size, the number of worker threads) and by dynamically refining sections of code already parallelized statically according to the characteristics of the actual application execution.

Before presenting the optimizations made at runtime, we detail the component responsible for executing jobs (i.e., the Workers Manager in Figure 5.8). Jobs are evenly distributed over workers. Each worker thread keeps a local queue of its slice of dispatched jobs and a circular buffer of transaction descriptors. A worker is in charge of executing transactions and

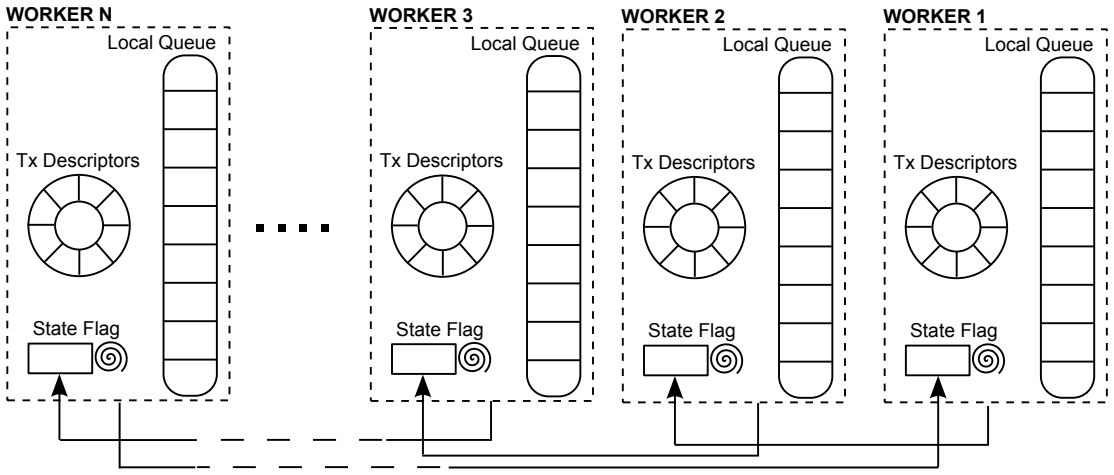


Figure 5.8: Workers Manager

keeping them in the *completed* state once they finish. As stated before, after the completion of a transaction, the worker can speculatively begin the next transaction. However, to avoid unmanaged behaviors, the number of speculative jobs is limited by the size of its circular buffer. The buffer size is crucial as it controls the lifetime of transactions. A larger buffer allows the worker to execute more transactions, but it increases also the transaction life time, and consequently the conflict probability.

The ordering is managed by a worker-local flag called *state flag*. This flag is read by the current worker, but is modified by its predecessor worker. Initially, only the first worker (executing the first job) has its state flag set, while others have their flag cleared. After completing the execution of each job, the worker checks its local state flag to determine if it is permitted to commit or proceed to the next transaction. If there are no more jobs to execute, or the transactions buffer is full, the worker spins on its state flag. Upon successful commit, the worker resets its flag and notifies its successor to commit its completed transactions. Finally, if one of the jobs has a break condition (i.e., not the *normal exit*) the workers manager stops other workers by setting their flags to a special value. This approach maximizes the use of cache locality as threads operate on their own transactions and access thread-local data structures, which also reduces bus contention.

5.8.1 Batch Size

The static analysis does not always provide information about the number of iterations, hence, we cannot accurately determine the best size for batching jobs. A large batch size may cause many aborts due to unreachable jobs, while having small batches increases the number of iterations between *dispatcher* and the *executor*, and, as a consequence, the number of pauses to perform due to Sync. In our implementation, we use an exponentially increasing batch size. Initially, we dispatch a single job, which covers the common set of loops with

zero iterations; if the loops are longer, then we increase the number of dispatched jobs exponentially until reaching a pre-configured threshold. Once a loop is entirely executed, we record the last batch size used so that, if the execution goes back on calling the same loop, we do not need to perform again the initial tuning.

5.8.2 Jobs Tiling and Partitioning

As explained in Section 5.5.2, the transformed program dispatches iterations as jobs, and our runtime runs jobs as transactions. Here we discuss an optimization, named *jobs tiling*, that allows the association of multiple jobs to a single transaction. Increasing jobs per transaction reduces the total number of commit operations. Also, it allows assigning enough computation power to the threads, which outweigh the cost of transactional setup. Nevertheless, tiling is a double-edged sword. Increasing tiles increases the size of read and write sets which can degrade performance. We tune tiling by taking into account the number of instructions per job, and commit rate of past executions using the *knowledge base*.

In contrast to tiling, a job may perform a considerable amount of non-transactional work. In this case, enclosing the whole job within the transaction boundaries makes the abort operation very costly. Instead, the transactifier pass checks the basic blocks with transactional operations and finds the nearest *common dominator* basic block for all of them. Given that, the transaction start (*tx_begin*) is moved to the common dominator block, and *tx_end* is placed at each *exit* basic block that is dominated by the common dominator. As a result, the job is partitioned into non-transactional work, which is now moved out of the transaction scope, and the transaction itself (See Figure 5.1c).

5.8.3 Workers Selection

Figure 5.3 shows how the *workers manager* module handles the concurrent executions. The number of worker threads in the pool is not fixed during the execution, and it can be changed by the *executor* module. The number of workers affects directly the transactional conflict probability. The smaller the number of concurrent workers, the lower the conflict probability. However, optimistically, increasing the number of workers can increase the overall parallelism (thus performance), and the underlying hardware utilization.

In practice, at the end of the execution of a batch of jobs, we calculate the throughput and we record it into the *knowledge base*, along with the commit rate, tiles and the number of workers involved. We apply a greedy strategy to find an effective number of workers by matching with the obtained throughput. The algorithm constructs a window of different worker counts, and iteratively improves it by changing the count of workers.

Using the throughput metric is better than relying on the commit rate. For example, three workers with an average commit rate equal to 50% are better than two workers with 70%

average commit rate. In addition to that, parallel execution is subject to other factors such as bus contention, cache hits, and thread overhead.The throughput combines all of these factors, thus giving a global picture about the performance gain.

Finally, in some situations (e.g., high contention or very small transactions) it is better to use a single worker (sequentially). For that reason, if our heuristic decides to use only one worker, then we use the non-transactional version (as a fast path) of the synthetic method to avoid the unnecessary transaction overhead.

5.8.4 Manual Tuning

As stated early, Lerna is completely automated but it still allows the programmer to provide hints about the program to parallelize. In this section we present some of the manual configurations that can be done. These configurations are only applicable if the source code is available.

A *safe call* is a call to an external function defined as a library or a system call; such a call must be stateless and cannot affect the program result if repeated multiple times. Such calls cannot be detected using static analysis, so we rely on the user for defining a list of them. Our framework generates a histogram of calls that represents the number of excluded blocks from our transformation because of this call. Based on this histogram, user can decide which calls are more beneficial to be classified as a safe call.

The alias analysis techniques (see Section 5.5.3) help in detecting dependencies between loads and stores; however, in some situations (as documented in [47]) it produces conservative decisions, which limit the opportunities of parallelization. It is non-trivial for the static analysis to detect aliases throughout nested calls. To assist the alias analysis, we try to inline the called functions within the transactional context. Nevertheless, it is common in many programs to find a function that does only loads of immutable variables (e.g., reading memory input). Marking such a function as read-only can significantly reduce the number of transactional reads, as we will be able to use the non-transactional version of the function, hence reducing the overall overhead.

Section 5.4 explains how Lerna detects the eligible code for parallelization through the profiling phase. Alternatively, users can directly inform our *Builder Pass* of their recommendations for applying our analysis. Also, the programmer can exclude some sections of the code from being parallelized for some reason. We support a user-defined *exclude list* for portions of code that will be excluded from any transformation.

	Configurations
Processor	2× Opteron 6168 processors
CPU Cores	12
Clock Speed	1.9 GHz
L1	128 KB
L2	512 KB
L3	12 MB
Memory	12 GB
OS	Ubuntu 10.04, Linux

Table 5.2: Testbed and platform parameters for Lerna experiments.

5.9 Evaluation

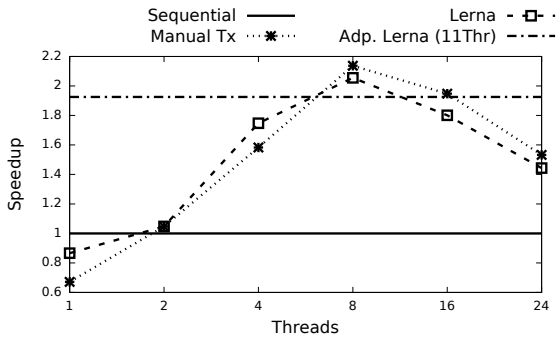
In this section we evaluate Lerna and measure the effect of the key performance parameters (e.g., job size, worker count, tiling) on the overall performance. Our evaluation involves a total of 13 applications grouped into micro-benchmarks and macro-benchmarks: STAMP [37] and PARSEC [133]. The micro-benchmarks allow us to tune the application workload in order to show strengths (and weaknesses) of our automated solution. The applications of the macro-benchmarks show the impact of Lerna in well-known workloads.

We compare the speedup of Lerna over the (original) sequential and the manual, optimized transactional version of the code (not available for PARSEC benchmarks). Note that the latter is not coded by us; it is released along with the benchmark itself, and is made manually by knowing the details of the application logic, thus it can leverage optimizations, such as the out-of-order commit, that cannot be caught by Lerna automatically. As a result, Lerna's performance goal is twofold: providing a substantial speedup over the sequential code, and to be as close as possible to the manual transactional version.

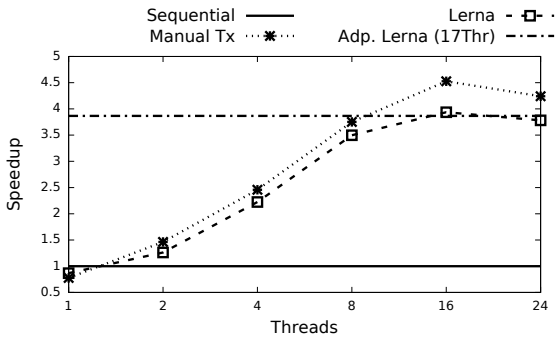
The testbed used in the evaluation consists of an AMD multicore machine equipped with 2 Opteron 6168 processors, each with 12-cores running at 1.9 GHz of clock speed. The total memory available is 12 GB and the cache sizes are 128 KB for the L1, 512 KB for the L2 and 12 MB for the L3. This machine represents well a current commodity hardware. On this machine, the overall refactoring process, from profiling to the generation of the binary, took ∼10s for micro-benchmarks and ∼40s for the macro-benchmarks. We did not use a higher core machine to avoid the effect of the Non-Uniform Memory Access (NUMA), where there is an additional latency according to the core used and the memory socket accessed. Optimizing TM for NUMA access is studied in details in [126].

5.9.1 Micro-benchmarks

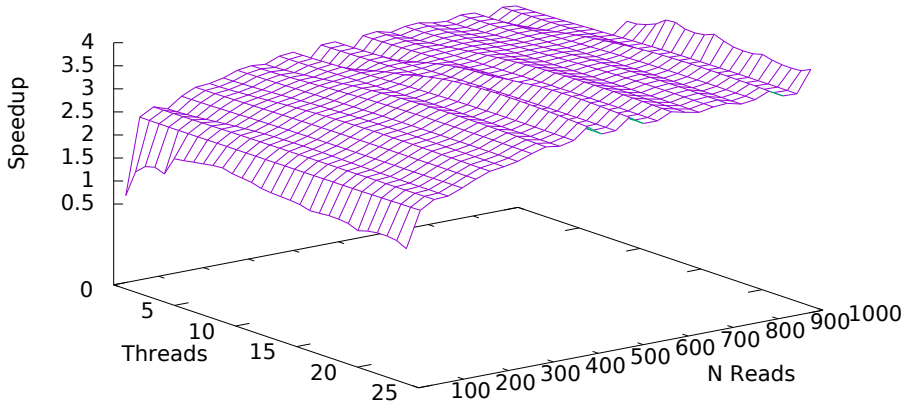
In our first set of experiments we consider the RSTM micro-benchmarks [2] to evaluate the effect of different workload characteristics, such as the amount of transactional operations



(a) Read100Write1 - Speedup



(b) Read1000Write1 - Speedup



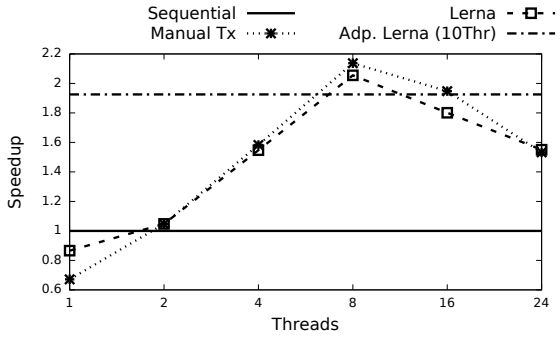
(c) Lerna Speedup

Figure 5.9: ReadNWrite1 Benchmark.

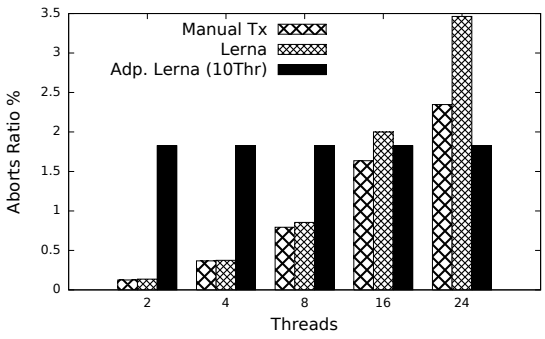
per job, the job length, and the read/write ratio, on the overall performance.

We report the speedup over the sequential code by varying the number of threads used. The performance is measured for two versions of Lerna: one adaptive, where the most effective number of workers is selected at runtime (thus its performance do not depend on the number of threads reported in the x-axis), and one with a fixed number of workers. We also reported the percentage of aborted transactions (right y-axis). As a general comment, we observe some small slow-down only when one thread is used; otherwise Lerna is usually very close to the manual transactional version. Our adaptive version gains on average 2.7× over the original code and it is effective because it finds (or is close to) the configuration where the top performance is reached.

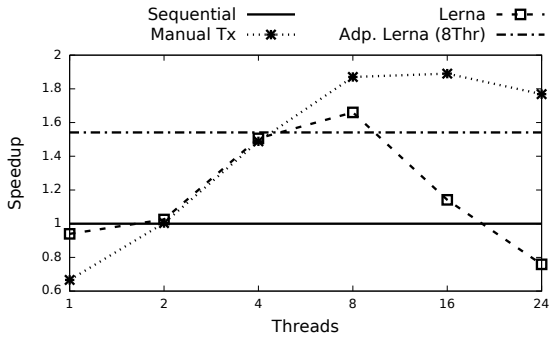
In *ReadNWrite1Bench* (Figure 5.9), transactions read N locations and write 1 location.



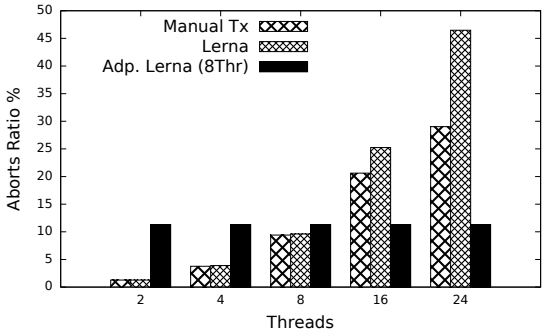
(a) ReadWrite100 - Speedup



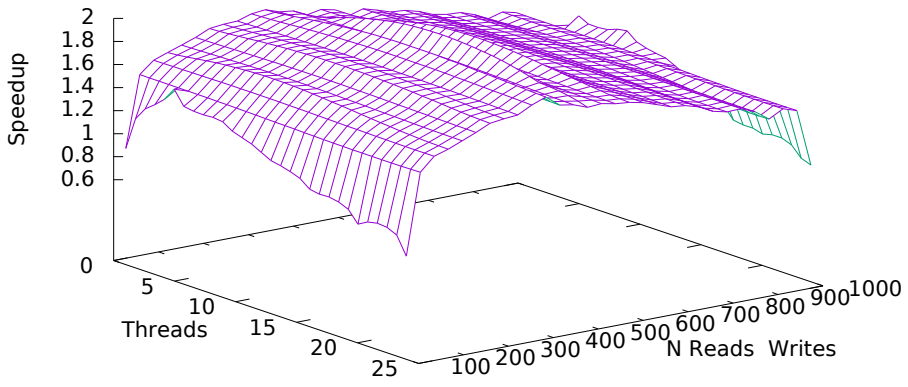
(b) ReadWrite100 - Aborts



(c) ReadWrite1000 - Speedup



(d) ReadWrite1000 - Aborts



(e) Lerna Speedup

Figure 5.10: ReadWriteN Benchmark.

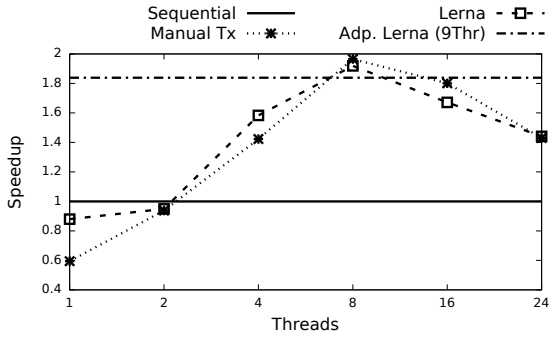
Given that, the transaction write-set is very small, hence it implies a fast commit of a lazy TM algorithm as ours. The abort rate is low (0% in the experiments), and the transaction length is proportional to the read-set size. Figure 5.9c illustrates how the size of transaction read-set (with a small size write-set) affects the speedup. Lerna performs closer to the manual Tx version; however, when transactions become smaller, the ordering overhead slightly outweighs the benefit of more parallel threads.

In *ReadWriteN* (Figure 5.10), each transaction reads N locations, and then writes to another N locations. The large transaction write-set introduces a delay at commit time for lazy versioning TM algorithms, and increases the number of aborts. Both Lerna and manual Tx incur performance degradation at high numbers of threads due to the high abort rate (up to 50%). In addition, for Lerna the commit phase of long transactions forces some (ready to commit) workers to wait for their predecessor, thus degrading the overall performance. In such scenarios, the adaptive worker selection helps Lerna avoid this degradation.

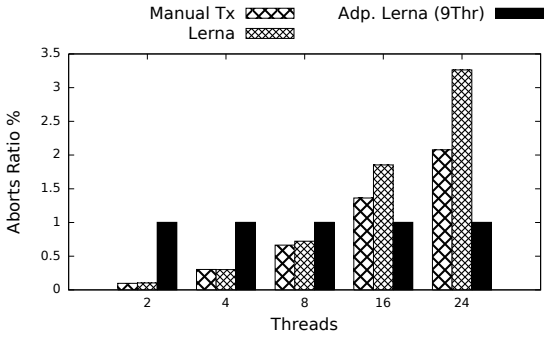
MCASBench performs a multi-word compare and swap, by reading and then writing N consecutive locations. Similarly to ReadWriteN, the write-set is large, but the abort probability is lower than before because each pair of read and write acts on the same location. Figure 5.11 illustrates the impact of increasing workers with long and short transactions. In Figure 5.11e, we see that with increasing the number of operations per transaction, the speedup degrades; besides threads contention, the number of aborts increases with increasing the number of accessed locations (See Figures 5.11b and 5.11d).

Interestingly, unlike the manual Tx, Lerna performs better at single thread because it uses the fast path version of the jobs (non-transactional) to avoid needless overhead. It worth noting that all micro-benchmarks does not perform many calculations on accessed memory locations, which represents a challenge for a TM-based approach.

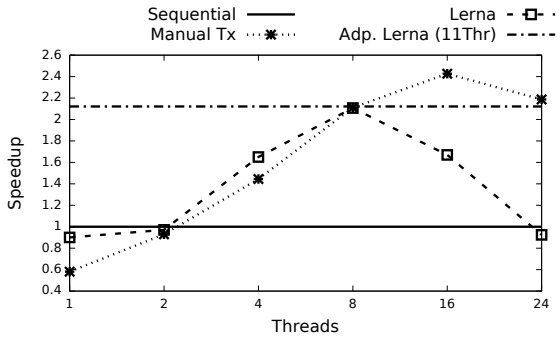
Figure 5.12 summarizes the behavior of the adaptive selection of the number of workers for the three micro-benchmarks and varying the size of the batch. The procedure starts by trying different worker counts within a fixed window (shown here, it is 7), then it picks the best worker according to the calculated throughput. Changing the worker counts shifts the window, thus allowing the technique to learn more and find the most effective settings for the current execution.



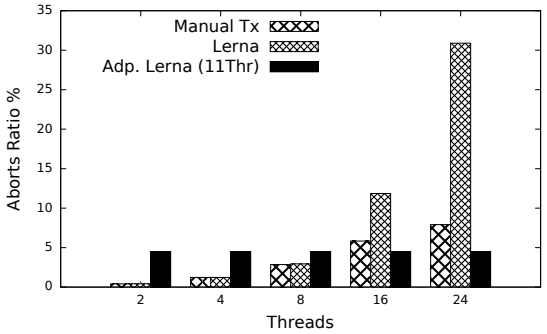
(a) MCAS100 - Speedup



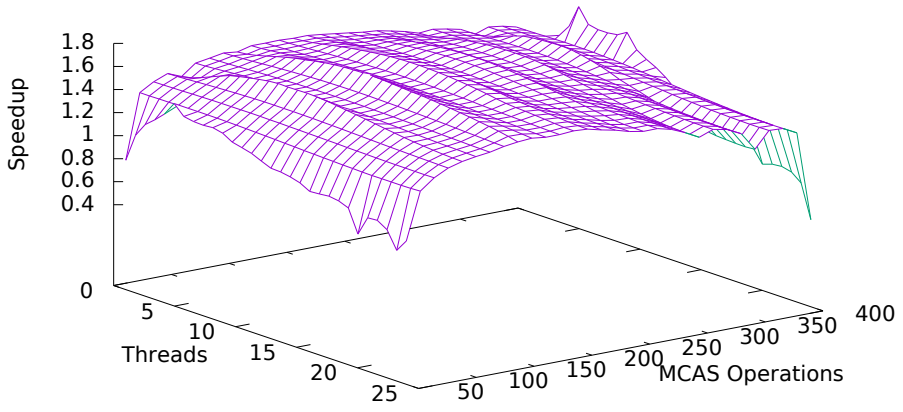
(b) MCAS100 - Aborts



(c) MCAS300 - Speedup



(d) MCAS300 - Aborts



(e) Lerna Speedup

Figure 5.11: MCAS Benchmark.

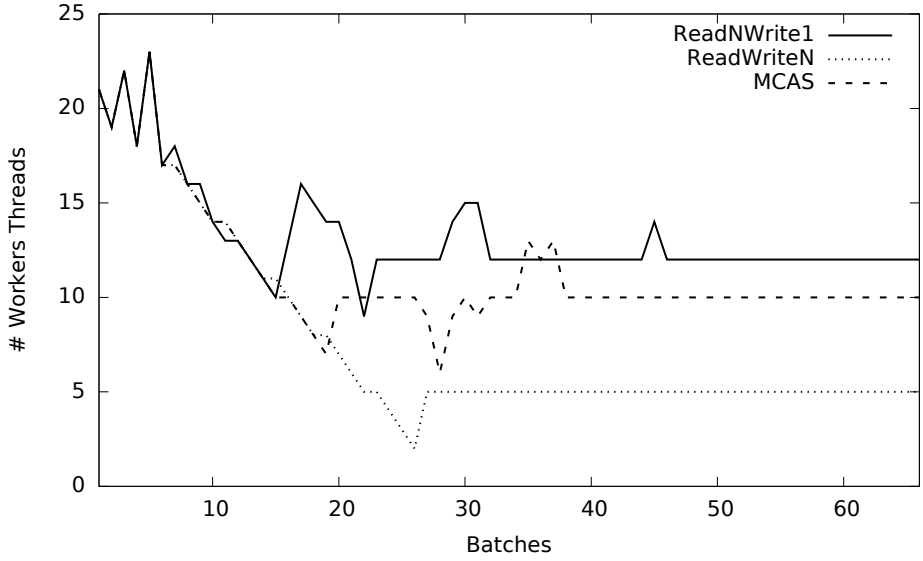


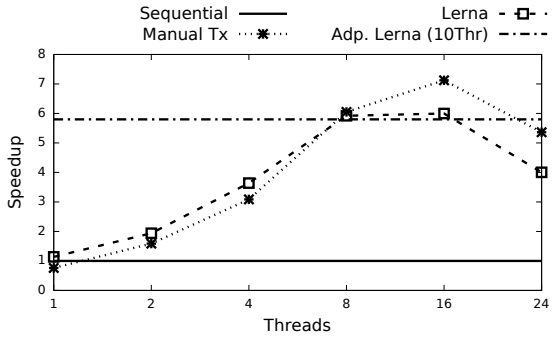
Figure 5.12: Adaptive workers selection

5.9.2 The STAMP Benchmark

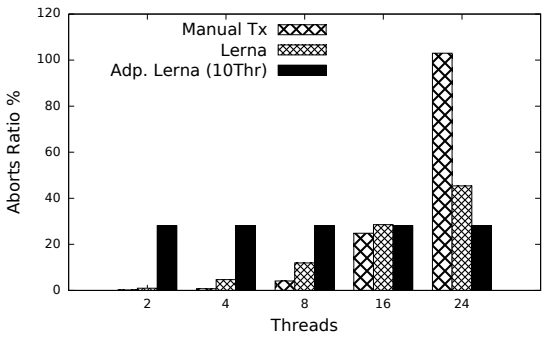
STAMP [37] is a comprehensive benchmark suite with eight applications that cover a variety of domains. Figures 5.13, 5.14 and 5.15 show the speedup of the Lerna's transformed code over sequential code, and against the manually transactified version of the application, which exploits unordered commit. While the main focus of our work is speedup over baseline sequential code, we highlight here the overheads and tradeoff with respect to a handcraft manual transformation aware of the underlying program semantics. Two applications (Yada and Bayes) have been excluded because they expose non-deterministic behaviors, thus their evolution is unpredictable when executed transactionally. Table 5.3 provides a summary of benchmarks and inputs used in the evaluation.

Kmeans, a clustering algorithm, iterates over a set of points and associate them to clusters. The main computation in finding nearest point, while shared data updates occur at the end of each iteration. Using job partitioning, Lerna achieves 6× and 1.6× speedup over the sequential version (See Figures 5.13a - 5.13d). The ordering introduces 25% delay compared to the unordered transactional version.

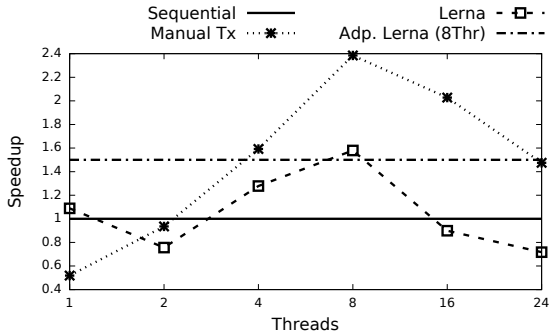
Genome, a gene sequencing program, reconstructs the gene sequence from segments of a larger gene. It uses a shared hash-table to organize the segments, which requires synchronization over its accesses. In Figures 5.13e - 5.13h, Lerna has 3× to 5.5× speedup over sequential. Ordering semantics is not a must for the hash-table insertion, which causes the manual Tx to perform 1.4× to 1.8× faster than Lerna, as transactions commit as soon as they end.



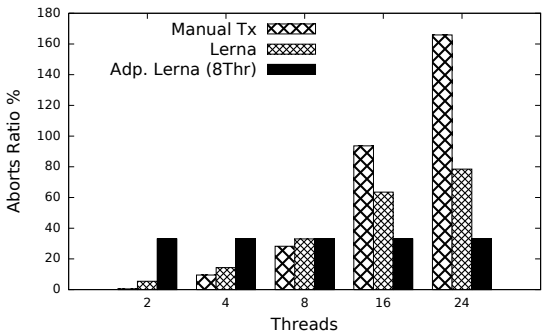
(a) Kmeans, Low Contention - Speedup



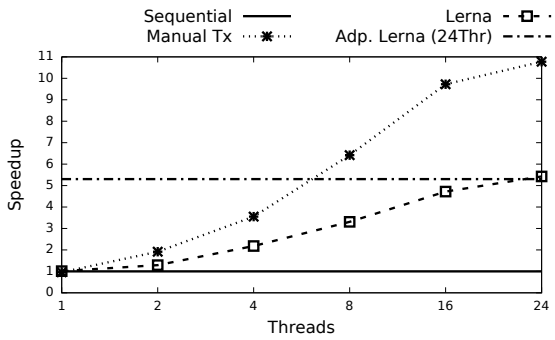
(b) Kmeans, Low Contention - Aborts



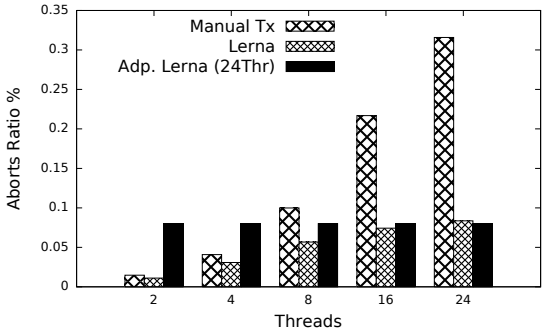
(c) Kmeans, High Contention - Speedup



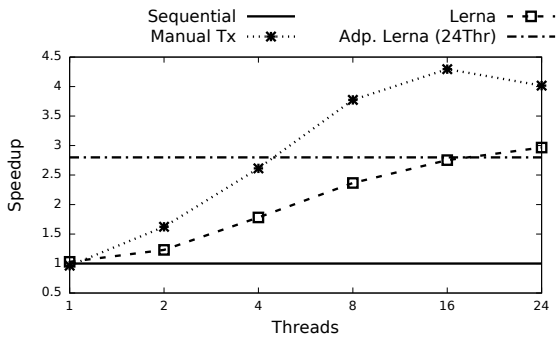
(d) Kmeans, High Contention - Aborts



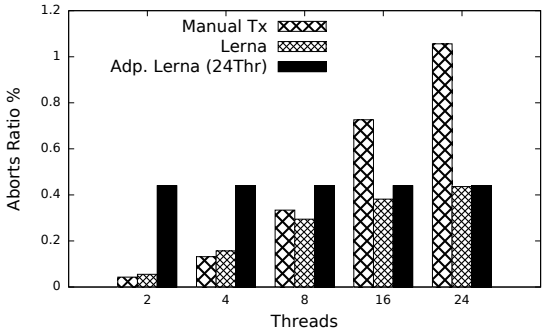
(e) Genome, Low Contention - Speedup



(f) Genome, Low Contention - Aborts



(g) Genome, High Contention - Speedup



(h) Genome, High Contention - Aborts

Figure 5.13: Kmeans and Genome Benchmarks

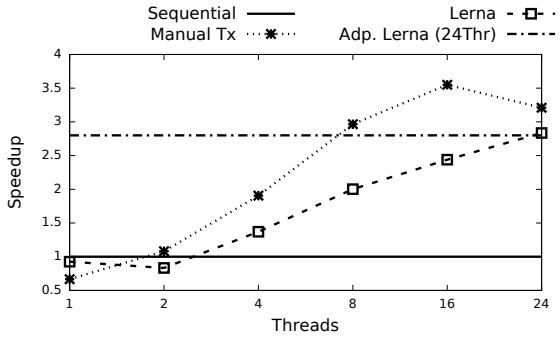
Benchmark	*Configurations*		Description
Kmeans	*Low*	-m60 -n60 -t0.00001 n65536 -d128 -c16	m: *max clusters*, n: *min clusters*
	High	-m20 -n20 -t0.00001 n65536 -d128 -c16	
Genome	*Low*	-g16384 -s64 -n86777216	n: *number of segments*
	High	-g16384 -s64 -n16777216	
Vacation	*Low*	-n30 -q90 -u100 -r1048576 -t4194304	n: *queries*, q: *relations queried ratio*
	High	-n50 -q60 -u100 -r1048576 -t4194304	
SSCA2	*Low*	-s20 -i0.1 -u0.1 -l3 -p3	s: *problem scale*, i: *inter cliques ratio*,
	High	-s19 -i0.5 -u0.5 -l3 -p3	u: *unidirectional ratio*
Labyrinth	*Low*	-x128 -y128 -z3 -n128	x, y, z: *maze dimensions*, n: *exits*
	High	-x32 -y32 -z3 -n96	
Intruder	*Low*	-a10 -l128 -n262144 -s1	l: *max number of packets per stream*
	High	-a10 -l12 -n262144 -s1	

Table 5.3: Input configurations for STAMP benchmarks.

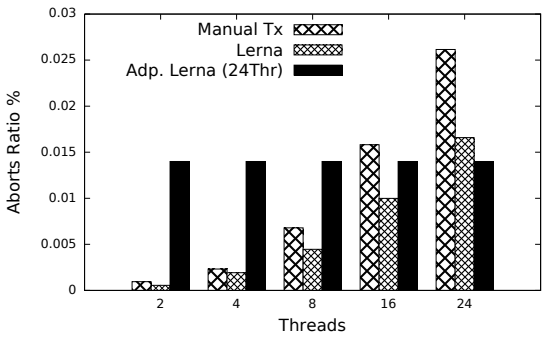
Vacation is a travel reservation system using an in-memory database. The workload consists of clients reservation. This application emulated an OLTP workload. Lerna improves the performance by 2.8× faster than the sequential system, and it is very close to the manual (See Figures 5.14a - 5.14d). Vacation transactions do not inhibit a lot of aborts as it accesses relatively large amount of data.

SCAA2 is a multi-graph kernel that is commonly used in domains such as biology and security. The core of the kernel uses a shared graph structure that is updated at each iteration. The transformed kernel outperforms the original by 1.4×, while dropping the in-order commit allows up to 3.9× (See Figures 5.14e - 5.14h).

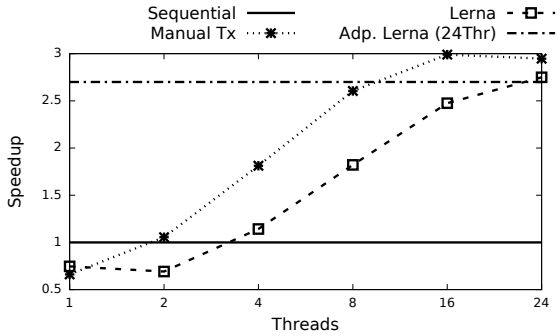
Lerna exhibits no speedup using *Labyrinth* and *Intruder* (See Figure 5.15) because they use an internal shared queue for storing the processed elements and they access it at the beginning of each iteration to dispatch them for the execution (i.e., a single contention point). While our jobs execute as a single transaction, the manual transactional version, creates multiple transactions per iteration. The first iteration, handle just the queue synchronization, while other iterations do the processing. Jobs partitioning will not help in this situation because the shared access occur at the beginning of the iteration. Assume splitting each job into two transactions, the ordering between jobs will prevent the first transaction at higher iterations from commit. However, we can see that this technique can help to parallelize two concurrent iterations at most; the first transaction at the higher index iteration can access the shared queue right after the corresponding lower index iteration commits.



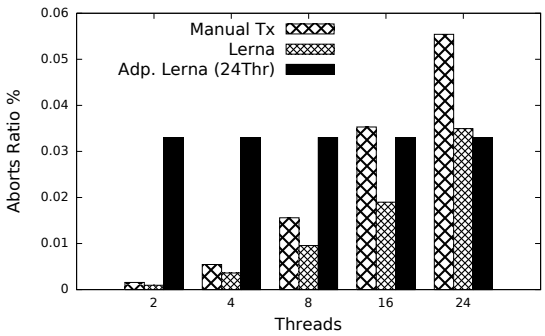
(a) Vacation, Low Contention - Speedup



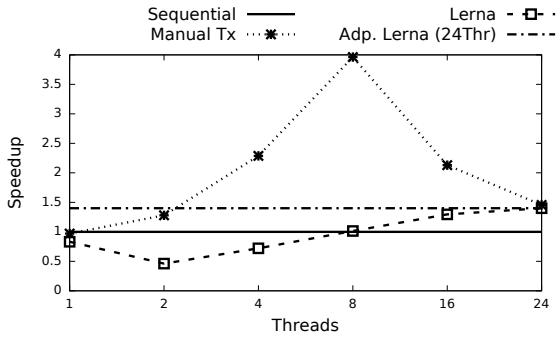
(b) Vacation, Low Contention - Aborts



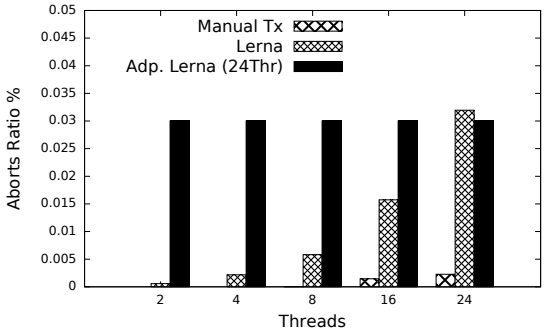
(c) Vacation, High Contention - Speedup



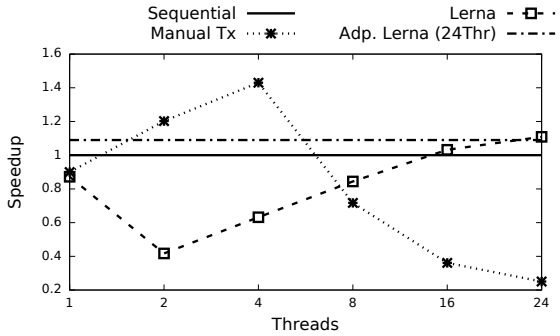
(d) Vacation, High Contention - Aborts



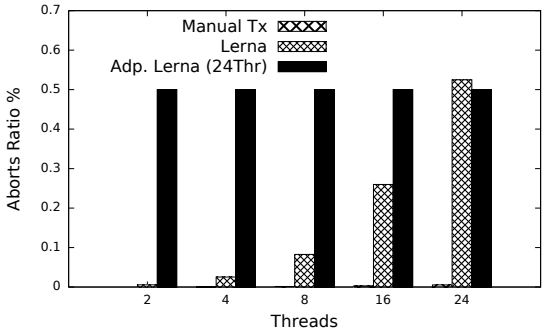
(e) SCAA2, Low Contention - Speedup



(f) SCAA2, Low Contention - Aborts

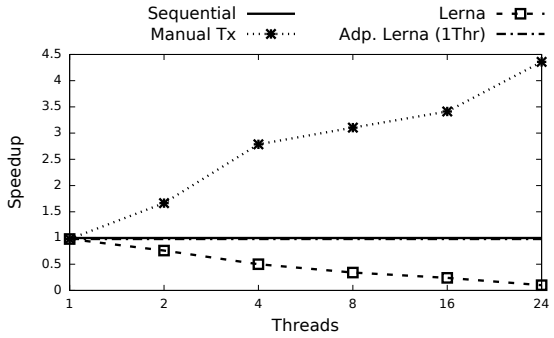


(g) SCAA2, High Contention - Speedup

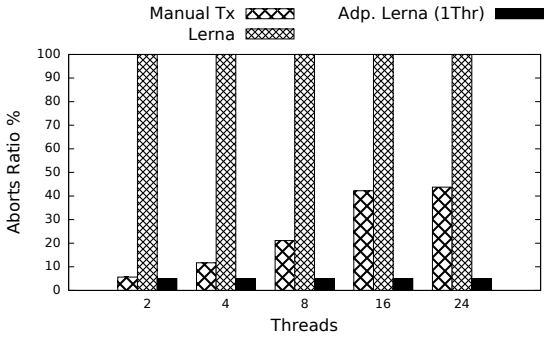


(h) SCAA2, High Contention - Aborts

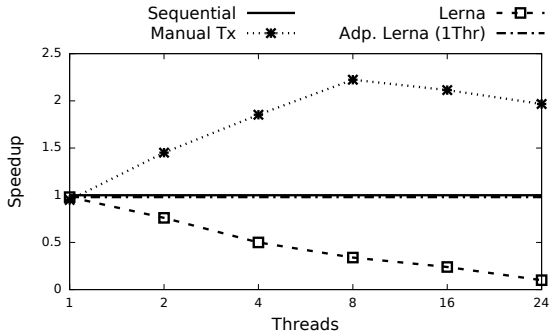
Figure 5.14: Vacation and SSCA2 Benchmarks



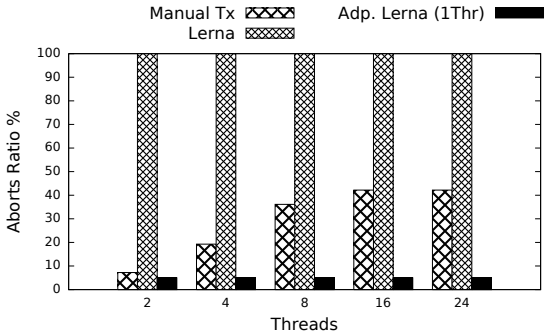
(a) Labyrinth, Low Contention - Speedup



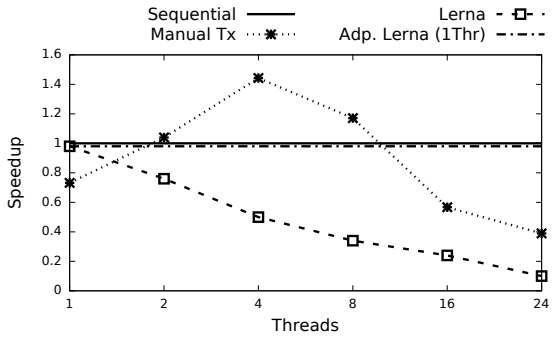
(b) Labyrinth, Low Contention - Aborts



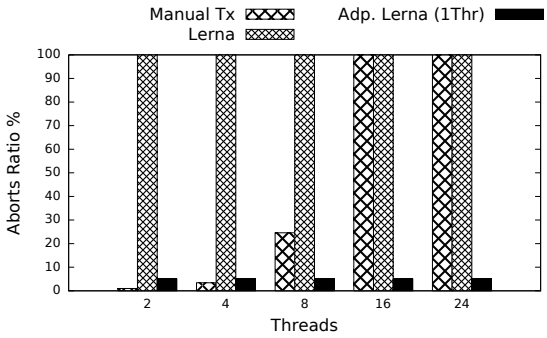
(c) Labyrinth, High Contention - Speedup



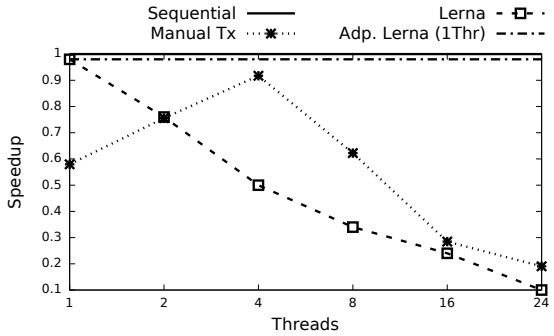
(d) Labyrinth, High Contention - Aborts



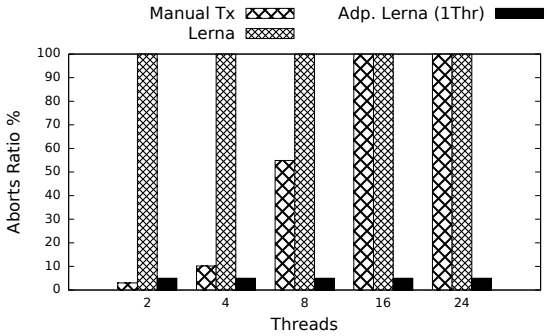
(e) Intruder, Low Contention - Speedup



(f) Intruder, Low Contention - Aborts



(g) Intruder, High Contention - Speedup



(h) Intruder, High Contention - Aborts

Figure 5.15: Labyrinth and Intruder Benchmarks

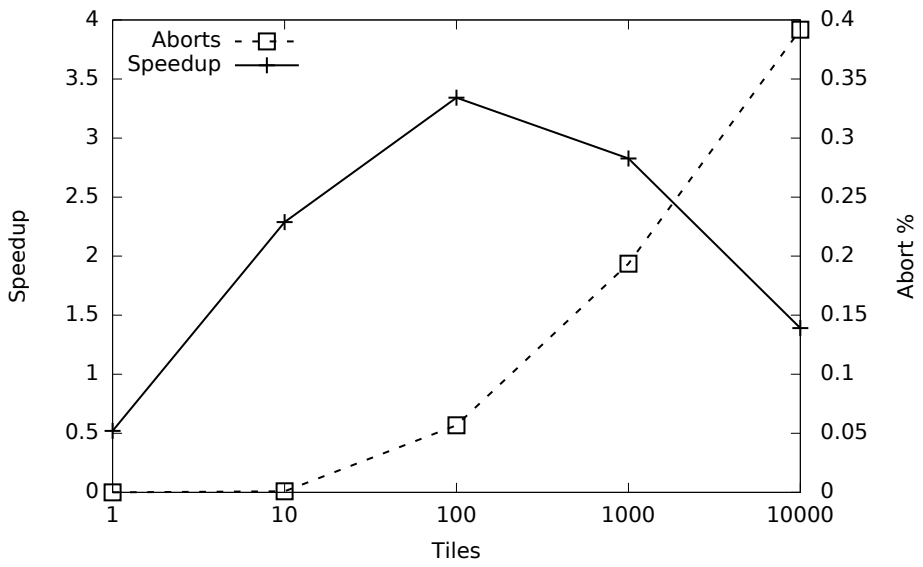
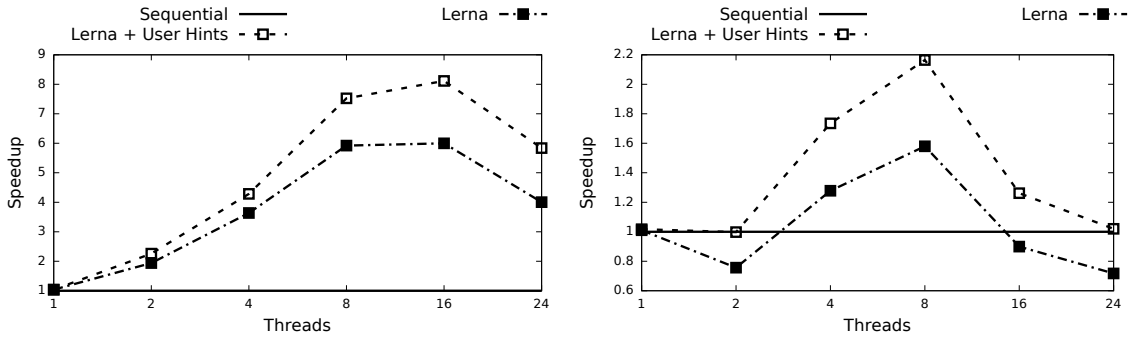


Figure 5.16: Effect of Tiling on abort and speedup using 8 workers and Genome.

As explained in Section 5.8.2, selecting the number of jobs per each transaction (jobs tiling) is crucial for performance. Figure 5.16 shows the speedup and abort rate with changing the number of jobs per transaction from 1 to 10000 using the Genome benchmark. Although the abort rate decreases when reducing the number of jobs per transaction, it does not achieve the best speedup. The reason is that the overhead for setting up transactions nullifies the gain of executing small jobs. For this reason, we dynamically set the job tiling according to the job size and the gathered throughput.

The manual tuning further assists Lerna for improving the code analysis and eliminating any avoidable overhead. An evidence of this is reported in Figure 5.17 where we show the speedup of Kmeans against different numbers of worker threads using two variants of the transformed



(a) Low Contention (b) High Contention

Figure 5.17: Kmeans performance with user intervention

Benchmark	*Configurations*	Description
Fluidanimate	-i in_500K	i: *input file with 500K particles data*
Swaptions	-ns 100000 -sm 1 -nt 1	ns: *number of swaptions*
Blackscholes	-i in_10M	i: *input file with 10 millions equations data*
Ferret	top=50 depth=5	top: *top K*, depth: *depth*

Table 5.4: Input configurations for PARSEC benchmarks.

code using Lerna: the first is the normal automatic transformation, and the second leverages user's hints about memory locations that can be accessed safely (i.e., non-transactionally). The figure shows that Lerna's transformed code outperforms even the manual transactional code.

5.9.3 The PARSEC Benchmark

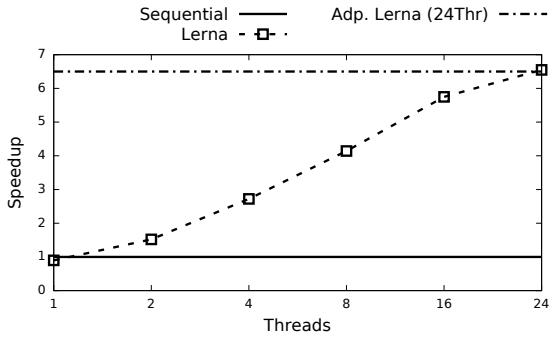
PARSEC [23] is a benchmark suite for shared memory chip-multiprocessors architectures. We evaluate Lerna performance using a subset of these benchmarks which cover different aspects of our implementation. Table 5.4 provides a summary of benchmarks and inputs used in the evaluation.

The Black-Scholes equation [100] is a differential equation that describes how, under a certain set of assumptions, the value of an option changes as the price of the underlying asset changes. This benchmark calculates Black-Scholes equation for input values and produces the results. The iterations are relatively short; which causes producing a lot of jobs in Lerna's transformed program. However, jobs can be tiled (See Section 5.8.2) where each group of iterations execute within a single job. Another approach is to add an unrolling pass earlier to our transformation. Figure 5.18c shows the speedup with different values for loop unrollings.

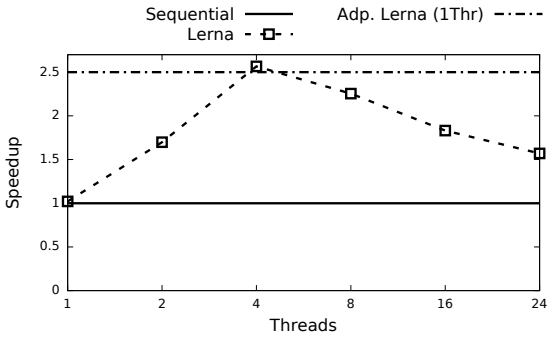
Swaptions benchmark contains routines to compute various security prices using Heath-Jarrow-Morton (HJM) [86] framework. Swaptions employs Monte Calro simulation (MC) to compute prices. Figure 5.18b shows Lerna speedup over the sequential code.

Fluidanimate [133] is a known application performing physics simulations (about incompressible fluids) to animate arbitrary fluid motion by using a particle-based approach. The main computation is spent on computing particle densities and forces, which involves six levels of loops nesting updating a shared array structure. However, iterations updates a global shared matrix of particles; which makes every concurrent transaction conflicts with its preceding transactions (See Figure 5.18d).

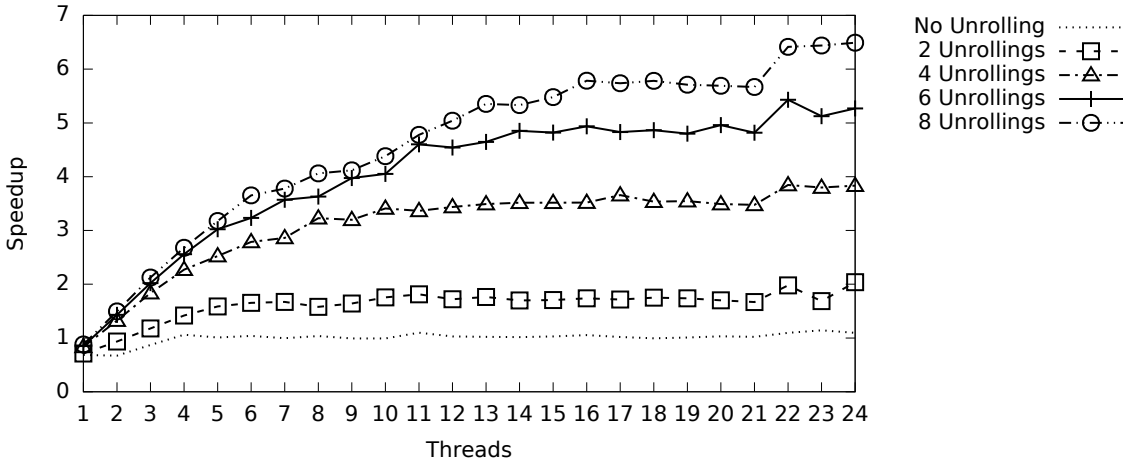
Ferret is a toolkit which is used for content-based similarity search. The benchmark workload is a set of queries for image similarity search. Similar to *Labyrinth* and *Intruder*, Ferret uses a shared queue to process its queries; which represents a single contention point and prevents any speedup with Lerna (See Figure 5.18e).



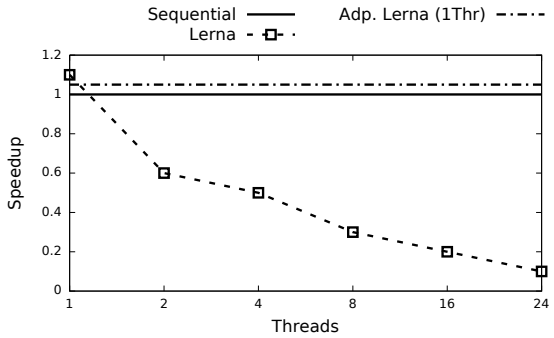
(a) Blackscholes



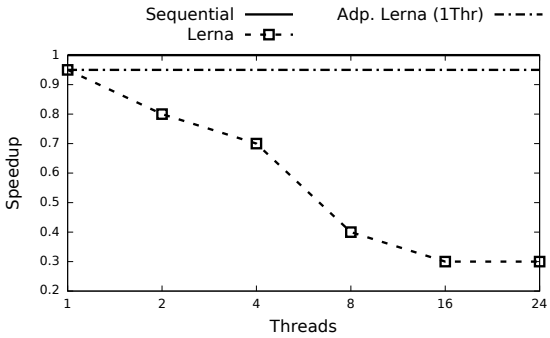
(b) Swaptions



(c) Effect of Loop Unrolling on speedup using Blackscholes



(d) Fluidanimate



(e) Ferret

Figure 5.18: PARSEC Benchmarks

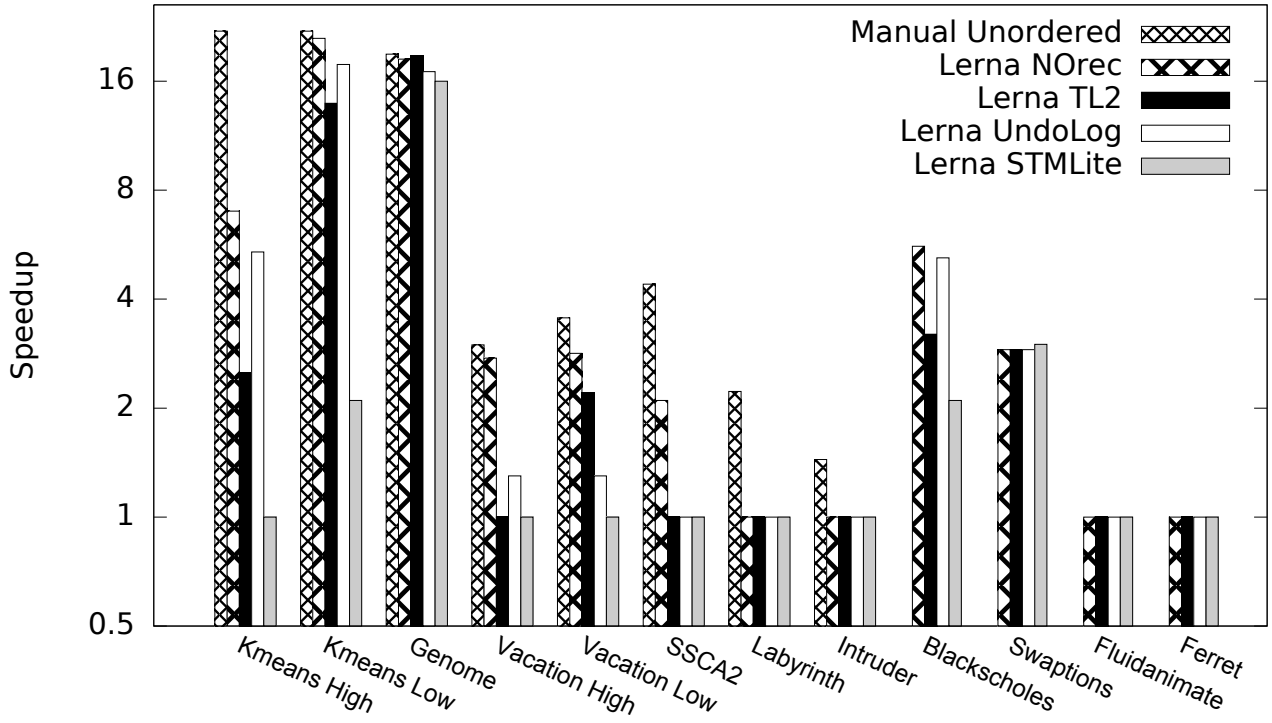


Figure 5.19: Effect of changing the TM algorithm (y-axis in log-scale)

5.9.4 The Effect of Changing TM Algorithm

Figure 5.19 shows the speedup of Lerna's transformed code over the sequential code, and against the manual transactional version of the applications, which exploits unordered transactions commits.[3]

In *Kmeans*, under high contention, NOrec is 3× slower compared to the manual unordered transactional version (more data conflicts and stalling overhead); however they are very close in the low contention scenario. TL2 and STMLite suffer from false conflicts (given the limited lock table or signatures) which limits their scalability.

Genome conducts a large number of read-only transactions (Hashtable *exists* operation); a friendly behavior for all implemented algorithms. TL2 is just 10% slower than the manual competitor. Similarly, *Swaptions* benefits from its small size write-set which produces a similar speedup with all TM algorithms integrated.

With Vacation, Lerna using NOrec achieves a peek performance of 2.8× faster than sequential, and it is very close to the manual. TL2 performance is dependent on the lock contention;

[3]In this experiment, we used different input configurations than the ones shown at Table 5.3 and Table 5.4 to illustrate the performance differences between the integrated TM algorithms.

2.2× under low contention, and no speedup under high contention.

The transformed *SSCA* kernel outperforms the original by 2.1× using NOrec, while dropping the in-order commit allows up to 4.4×. It worth noting that NOrec is the only algorithm that manage to achieve speedup because it tolerates high contention and isn't affected by false sharing as it deploys a value-based validation.

In *Black-Scholes*, the iterations are relatively short. Algorithms with low overhead, such as NOrec and UndoLog, outperform others: TL2 and STMLite. All the TM algorithms produce speedup between 2.1× and 5.6×.

As discussed before in Sections 5.9.2 and 5.9.3, regardless of the underlying TM algorithm, Lerna exhibits no speedup using *Labyrinth*, *Intruder*, *Fluidanimate* and *Ferret*.

5.10 Discussion

Lerna can be applied to all applications other than the used benchmarks. Here we discuss the lesson learnt from our evaluation in order to provide an intuition about which are the (negative) cases where Lerna's parallelization refactoring is less effective.

Lerna extracts parallelism when possible. There are scenarios where, without the programmer handing the application's logic on the refactoring process, Lerna encounters some hindrance (e.g., single point of contention) that cannot automatically break due to the lack of "semantic" knowledge. Examples of that include complex data structure operations. Examples of that include *Labyrinth*, *Intruder* and *Ferret* as explained before, or data-level conflicts as in *Fluidanimate*. We also looked into SPEC [88] applications, and we found that most of them use data structures iterators.

The primary factors of overhead are: ordering transactions, contention on accessing shared data (e.g., implied constraint by underlying bus-based architecture), and aborting conflicting transactions because of true data or control dependencies, or false conflicts.

In addition, Lerna becomes less effective when: there are loops with few iterations (e.g., Fluidanimate) because the actual application parallelization degree is limited; there is an irreducible global access at the beginning of each loop iteration, thus increasing the chance of invalidating most transactions from the very beginning (e.g., Labyrinth); and workload is heavily unbalanced across iterations. Anyway, in all the above cases, at worst, the code produced by Lerna performs as the original.

Chapter 6

Ordered Write Back Algorithm

Ordering transactions before the execution is a known problem, mostly relevant to deployments where an external service is in charge of providing the commit order to satisfy certain properties (e.g., equivalent semantics or system dependability). Examples of these deployments include (but are not limited to): loop parallelization [178, 73, 184, 160], and fault-tolerance using the state machine replication (SMR) approach [169, 115]. In the former, loops designed to run sequentially are parallelized by executing their iterations concurrently, and guarded by TM transactions, as in [73, 160], to handle conflicts (i.e., data dependency) correctly. In that case, providing an order matching the sequential one is fundamental to enforce a semantic (of the parallel code) that is equivalent to the original (sequential) code. Regarding the latter, SMR-based transactional systems order transactions (totally or partially) before their execution to guarantee that a state always evolves on several computing nodes, consistently. In order to achieve this, usually a consensus protocol is employed (e.g., Paxos [106]), which establishes a common order among transactions; this order must be then enforced while processing and committing those transactions.

In this and the next chapters, we present two Software TM implementations that outperform all baseline and specialized solutions that commit transactions in-order (e.g., STMLite [120]): *Ordered Write Back* (OWB) and *Ordered Undo Logging* (OUL) [164]. They are based on two widely used techniques to merge transactions modifications into the shared state, namely write back (in OWB) and write through (in OUL). Both OWB and OUL deploy a common design that uses a *dependency-aware cooperative model*: transactions employ a weaker isolation level, and exchange both data and locks to increase concurrency while preserving the commit order. More specifically, OWB uses data forwarding for transactions that finish their execution successfully, but are not committed yet, and OUL leverages encounter time locking with the ability to pass the lock ownership to other transactions. We also provide a variant of OUL (OUL-Steal) that deploys a lock-stealing technique.

Given its design principle, which allows transactions to access modifications made by committing transactions that may abort later due to disorderly executions, OWB is safe as it

ensures TMS1 [62], a weaker consistency condition than opacity [76, 77], the most popular consistency condition for TM. TMS1 has been proved to be sufficient to guarantee safety in our model [13], as is the case with opacity. OUL achieves higher concurrency and therefore higher performance, at the cost of weakening the correctness level by ensuring Strict Serializability [21].

In this chapter, we present an analysis that identifies the major bottlenecks in processing transactions in parallel while guaranteeing a pre-defined commit order (Section 6.3). Next, we propose a methodology that effectively solves this problem by combining dependency awareness and locks forwarding with undo-log techniques, which overcomes competitors' drawbacks (Section 6.4). Finally, we present OWB, which is, to the best of our knowledge, the first TM implementation[1] that satisfies TMS1.

6.1 Commit Order

In Chapters 4 and 5, we exploit Transactional memory (TM) as a technique for protecting speculative code [93, 120], and extracting parallelization from sequential code [178, 73, 184, 160]. A conflict is handled by aborting (and re-executing) the transaction which ran code with the later chronological ordering. The key idea is that code blocks run as transactions and commit in the program's chronological order. The techniques for supporting the aforementioned ordering are classified as: blocking [73, 120, 185] or freezing [190] techniques. These techniques are described in detail in Section 6.3.

Using blocking, the thread executing the transaction with the lowest age commits. While a thread is committing, other threads wait until the commit completes. Upon a successful commit, the subsequent thread with the lowest age performs the commit. As an example of the blocking approach, Mehrara *et al.* [120] proposed a TM with a separate thread, the Transaction Commit Manager (TCM), that detects conflicts among transactions waiting for their turn to commit. TCM orchestrates the in-order commit process with the ability to have concurrent commits. Workers threads poll and stall to wait for the TCM's permission. Gonzalez *et al.* [73] use a distributed approach for handling the commitment order. Each thread employs a bounded circular buffer to store its completed transactions. If all buffer slots are exhausted, the thread stalls until one of the pending executed transactions reaches the correct commit order. TCC [81] is a hardware model for supporting transactions ordering. The key idea is employing *local-buffering*, using the processors cache, for keeping transaction changes, then wait for its order and flush (commit) its buffer if it is still valid.

Another existing solution lets threads *freeze* completed transactions and proceed to execute the higher age transactions, with the disadvantage of increasing the transaction lifetime (hence, a higher conflict probability). Zhang *et al.* [190] introduced this technique to support

[1] Such a TM is released as an open-source project at: `https://bitbucket.org/mohamed-m-saad/ordertm`

a pre-determined total order of transactions. A *next-to-commit* shared variable is used to preserve this order.

The level of atomicity could be an orthogonal classification for the aforementioned techniques. The classical TM model mandates transactions to see only *committed* values. However, concurrent transactions can *cooperate* and construct a *dependency graph* of uncommitted, yet exposed, values. Based on this graph, the transactions commit in the constructed order. Although this approach defines the inverse problem of the in-order commit, it is fruitful for us to exploit that. Ramadan *et al.* [143] proposed a cooperative approach for executing transactions using a Dependency Aware STM model (DASTM). In DASTM, transactions *forward* their uncommitted changes, and based on that, the runtime defines the commit order. Interestingly, transactions never abort (only stall their commit) – except in the case of deadlock. Both OWB and OUL algorithms employ data forwarding to permit read after write operations, and they do not suffer from the high overhead of maintaining the conflict graph.

Deterministic execution of TM may be seen as a distant related topic. Recently, in [147] it has been proposed an STM implementation that improves performance in case of deterministic execution. Deterministic execution is meant for reducing the possible parallelism in the system, whereas our approaches aim at introducing parallelism when a specific commit order is enforced.

6.2 Execution and Memory Model

A transaction is a unit of work that executes atomically (i.e., all or nothing) and in isolation from the concurrent work. Memory transactions run optimistically, and thus in parallel, even if they may access the same shared objects. However, when a conflict is detected, one of the transactions is aborted and restarted. Our model assumes a set of transactions $T = \{T_1, T_2, \ldots, T_N\}$. Transactions access shared objects using read and write operations, with their usual meaning [93]. We denote the sets of shared objects accessed by transaction T_k for read and write as `read-set(`T_k`)` and `write-set(`T_k`)`, respectively.

A transaction execution is defined as a sequence of operations, where each operation is represented by a pair of *invoke* and *return* events. Besides the read and write operations, whose semantics is the usual one, it also includes a commit operation that starts by invoking the *try-commit* event, whose return value is either *commit* or *abort*. Note that a transaction can also be aborted before invoking the try-commit event. A transaction that begins its execution and did not invoke the try-commit event yet is called *live*. A transaction that invoked the try-commit event but did not committed or aborted yet is called *commit-pending*. When a transaction is categorized as committed, it means that all its write operations have been executed permanently on the shared state; and when it is categorized as aborted, it means that its operations have no permanent effect. In both the cases, all meta-data is

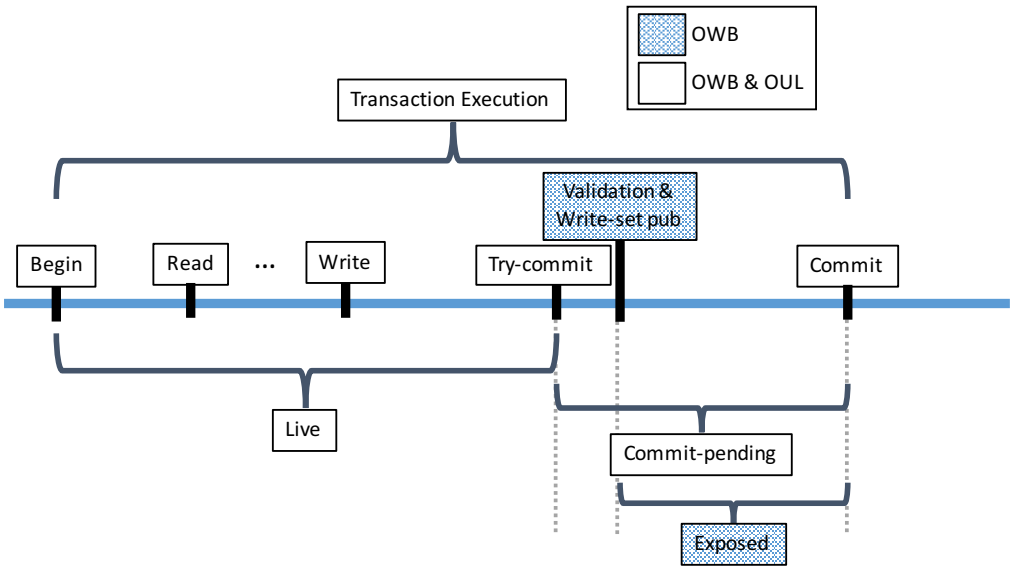


Figure 6.1: States of a transaction execution in OWB and OUL.

cleaned before proceeding or re-executing. Figure 6.1 summarizes the transaction states.

A shared object has a value and a (versioned) lock associated with it. We say that a shared object is *exposed* if it is locked by some live or commit-pending transaction. Intuitively, a shared object is exposed if some other transaction can already read it although its creator is still executing. We call the state of a transaction that is commit-pending and has all its written objects exposed as *exposed*.

Two transactions are said to be *conflicting* on an object X, if both are concurrent (i.e, non-committed or non-aborted yet) and access an object X, and at least one of them writes on X. Note that two transactions are conflicting even if both write the same object without reading it. Tracking such a dependency is fundamental, as motivated in the next paragraph. A conflict is handled by aborting one of the transactions, or postponing the access generating the conflict (if possible), until the other transaction commits.

6.2.1 Age-based Commit Order (ACO)

In this chapter, we focus on TM implementations providing a specific order of transaction commits, which is assumed to be known prior to the transaction execution. We denote such an order as the transaction *age*. The *age* of a transaction should not be confused with the transaction timestamp taken from a global timestamp, as used by many existing TM implementations (e.g., Transactional Locking 2 Algorithm (TL2) [59], Lazy Snapshot Algorithm (LSA) [150]). That timestamp is defined by the concurrency control according

to the actual state of execution and not externally (e.g., by the application) as age. In fact, in those solutions the transaction timestamp is leveraged to efficiently capture the modifications which happened to the shared state after the transaction's starting time. An illustrative distinction between age and such a timestamp is that when a transaction aborts, its timestamp is updated before retrying (as in TL2); the age cannot be updated.

The age is assumed to be defined by external factors before activating any transaction (e.g., an ordering layer deployed on top of the TM implementation), and must match the transactions commit order. Assigning ages to transactions establishes a *total order* on their commits. Let $\prec$ be the relation representing the total order on transaction ages, and let those ages be denoted as subscripts (e.g., T_x). An aborted transaction is restarted with the same value of age.

A concurrency control that enforces an order of commits ensures that when two operations o_i and o_j, issued by transactions T_i and T_j, respectively, are conflicting, then o_i must happen before o_j if and only if $T_i \prec T_j$ (i.e., i precedes j) [148]. We deploy this idea into our execution model by introducing an Age-based Commit Order (ACO) – also known as timestamp-based Commitment Ordering (TCO) in [148]. ACO mandates a customization of the classical TM model. As an example, the transaction conflict detection should guarantee that when $T_i \prec T_j$, T_i must not read a value written by T_j. We define a transaction T_j as *reachable* if all lower age transactions T_i, with $i < j$, are committed. This term depicts the fact that T_j has been reached by a serial execution where all transactions T_1,...,T_i committed in the order $\{1, \ldots, i, j\}$. Note that every transaction eventually commits in our model. This is due to the fact that as soon as a transaction aborts, it is restarted by the TM library.

6.3 Analyzing the Ordering Overhead

In this section, we analyze the overhead introduced by deploying a concurrency control that enforces a specific and pre-defined commit order. Assume an ACO where transaction T_i precedes transaction T_j ($T_i \prec T_j$). Although T_j may finish executing its operations before T_i, it must wait for T_i to commit first; then T_j can start committing (and validating if needed) its changes. This strategy forces the thread executing transactions with higher age to either: *block* [73, 120, 185] or *freeze* [190]. In both the strategies, an additional delay for the higher age transactions is introduced.

In the *blocking* approach, the thread spins waiting for the correct commit order [185], while the *freeze* approach allows threads to execute higher ordered transactions, and periodically check the commit-pending lower age transactions. In both the cases, the overall utilization is negatively affected: either by wasting processing cycles in waiting (stalling), or through re-executing the aborted transactions (as a result of increasing the conflict probability). Also, in the freeze approach, the overall performance is limited by the sum of the time consumed in performing commits, as they cannot trivially run in parallel. This is why in Figure 6.3, each

set of green rectangles (commits) is synchronized with the next batch of green rectangles (execution).

In the rest following, we show analytically the stalling periods (which we name *ord-delay* and indicate as Σ_δ) experienced by transactions in these techniques given the ordering constraints, using the best-case scenario, when there are no aborts among concurrent transactions. For simplicity, we assume that all transactions execute code equally long, and that the total number of active threads is equal to the number of cores to avoid the overhead of scheduling executions.

Let α_k be the time taken by T_k to execute all its operations when there are no conflicts. Let β_k be the time taken by T_k to commit its changes to the shared memory and to notify its successor transaction about the completion of the commit. Let δ_k be the idle time of the thread executing T_k before starting doing any useful work, such as committing T_k or executing T_{k+n}.

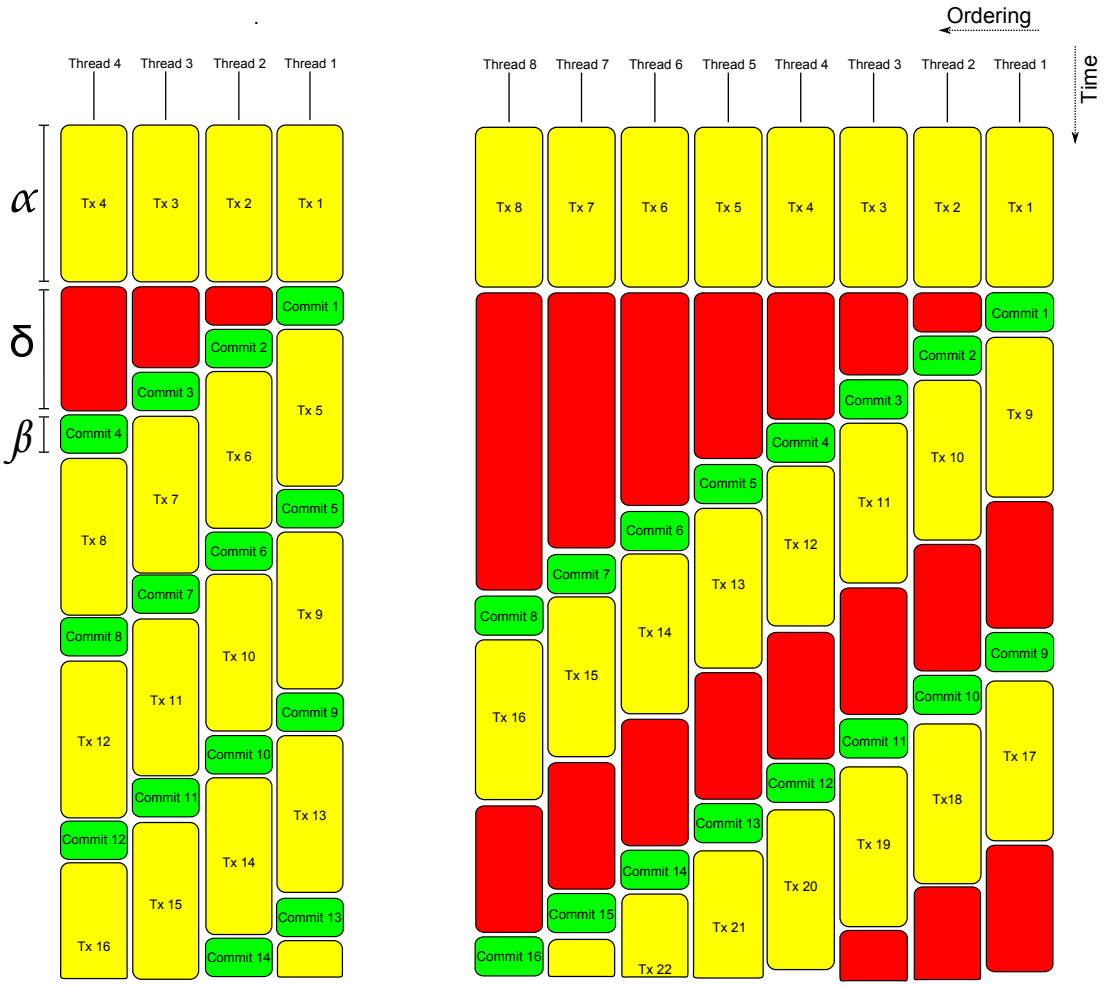
6.3.1 Blocking/Stall Approach

Assuming T_k is the last committed transaction, and transactions $T_{k+1}, \ldots, T_{k+n}$ are running in parallel on n processors. Given two transactions T_{k+i} and T_{k+j}, where $i < j \leq n$, if T_{k+j} completes its execution before T_{k+i} finalizes its commit, then T_{k+j} has to wait for δ_{k+j} time. The following expression shows the total wait time for N transactions distributed over n processors, and therefore activated at most n in parallel.

$$\Sigma_\delta = (N - n) * ((n - 1) * \beta - \alpha) + \beta * n * (n - 1)/2$$

The first n transactions (i.e., $T_1, \ldots, T_n$) start at the same time but they commit in order, causing the delay captured by the second term in the equation. For all other transactions (i.e., $T_{n+1}, \ldots, T_N$), although they have to wait for all their predecessors to commit, there could be an overlap between the transaction execution period and the commit of its predecessors. This (possible) delay is included in the first term of the formula.

Figures 6.2a and 6.2b show the execution of ordered transactions running over 4 and 8 threads respectively, where $\beta = 0.25\ \alpha$. Assume the transactions are distributed evenly over worker threads in round-robin, and the rightmost thread executes the lowest age transactions. With 4 threads, the stall occurs only at the beginning (the second term of the above equation), while 8 threads suffer from a continuous delay along the execution. Using the above formula, we can see that up to 5 threads, there is no continuous delay during the execution. The total ord-delay exhibits quadratic growth with an increasing number of threads, and it is affected by the ratio between the commit time (i.e., β) and the execution time (i.e., α).



(a) 4 Threads (b) 8 Threads

Figure 6.2: The Execution of Ordered Transactions using Blocking/Stall Approach

6.3.2 Freeze/Hold Approach

In contrast to the blocking approach, in the freeze approach, a thread transitions finished transactions into a freeze (hold) state [190], and executes their *commit* phases later on when all its preceding transactions (i.e., those with lower ages) become *reachable* (see Figure 6.3). Without loss of generality, we assume that a designated thread performs all the "frozen" commit phases. Similar to the blocking approach, we observe that threads stall to enable the committer thread. The following expression shows the ord-delay in case of N transactions.

$$\Sigma_{\delta} = \alpha + max(0, N * \beta - \alpha * \lceil (N - n)/n \rceil)$$

Note that, similarly to the blocking approach, the total delay is affected by the ratio between the commit time (β) and the execution time (α). However, the delay is relatively

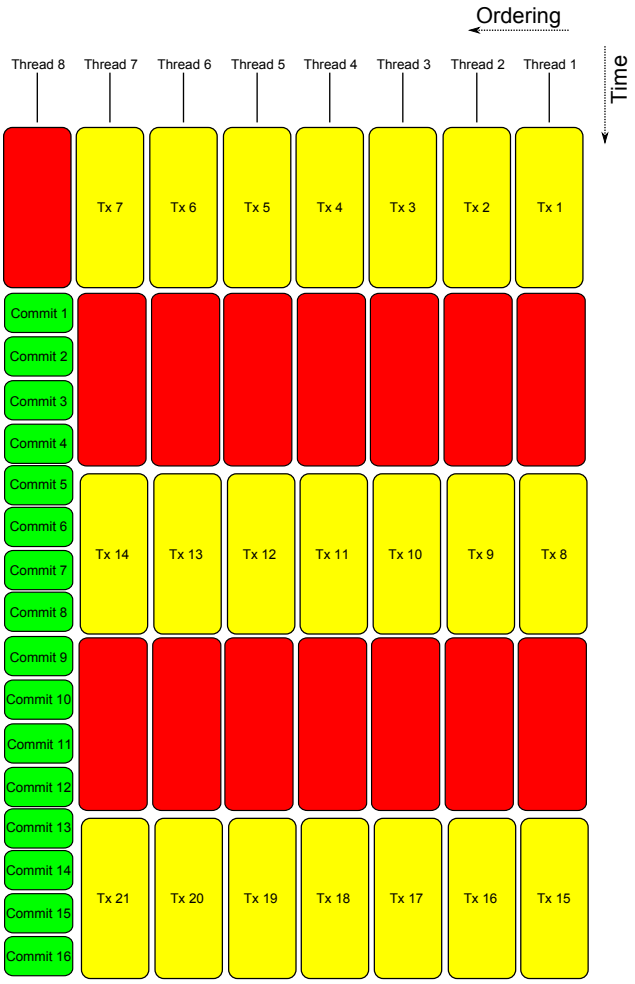


Figure 6.3: The Execution of Ordered Transactions using Freeze/Hold Approach

smaller than the blocking approach, and the number of live transactions doubled, increasing the probability of experiencing a conflict given the extended transaction lifetime (locks on accessed objects are held for longer).

Analyzing the results illustrated in this section, we can conclude that ordering transactions' commits negatively affects the overall resource utilization and may nullify any potential gain due to threads' parallelism.

In the following sections, we detail our proposals to overcome the above drawbacks, which introduce a new execution model and algorithms to reduce the ordering delay.

6.4 General Design

In this section, we present our co-operative model for supporting ACO. The most important factor that affects the transaction ord-delay, analyzed in the previous section, is the commit latency. In fact, the time required to execute the commit phase sets a lower bound on the ord-delay, while deferring the commit phase to a later point in time allows for more conflicts.

The core idea is to relax the common practice of letting transactions access values written by only committed or commit-pending transactions that will surely commit. In our proposed solutions, we weaken this assumption while still preserving the consistency according to ACO. Depending on the desired correctness and performance level, we permit a transaction to expose its changes either:

- after it invokes the try-commit event and performs a validation to make sure its execution is consistent, but still allowing it to abort later due to ACO violation (in OWB); or
- right after the write operation takes place during the execution and aborting any dependent transaction as soon as a further modification on the same early exposed value happens (in OUL).

The above idea allows transactions with higher age to use such visible changes. Although it speeds up the flow of values from lower age transactions to the higher age ones, it also creates a possible dependency chain with other live and commit-pending transactions that accessed those values. Therefore when an abort occurs, the abort event should be immediately triggered to all the dependent transactions (cascading abort).

Let T_1 and T_2 be concurrent transactions, where $T_1 \prec T_2$, and let X be a shared object accessed by both the transactions using write (W_i) and read (R_i) operations, where the suffix refers to the transaction Id. Let the operator $\rightarrow$ indicate the real-time order for the operations on X. In Table 6.1, we identify three different transactional models to handle concurrent read and write operations: the classical, the cooperative ordered (our proposed model), and the conflict serialization model (DASTM [143]).

The classical transactional model prohibits the co-existence of read and write operations on the same object issued by concurrent transactions, while the conflict serialization permits all combinations and enforces the commit order based on the transaction dependency. In contrast, the cooperative ordered model restricts the memory snapshot seen by current transactions to only the values exposed by older transactions. Under the conflict serialization model, transactions are aborted only when a mutual dependency (i.e., a cycle in the transaction dependency graph) exists; while in ACO, the graph is necessarily acyclic.

6.4.1 Cooperative Ordered Transactional Execution

To construct our cooperative model, we start by highlighting the following two events of a transaction execution.

- A transaction becomes *exposed*, when all its written objects have been exposed, and it is in the commit-pending state by having all its read operations consistent according to the ACO, therefore no conflict with any lower age transaction occurred.
- A transaction becomes *reachable*, when all the lower age transactions have been committed.

Exposing written objects before being sure that a transaction eventually commits may violate the ACO if all lower age transactions are not committed yet, or when the transaction conflicts with a lower age transaction that did not run at the time of becoming exposed. For this reason, we postpone releasing the transaction meta-data until the transaction becomes reachable, thus providing a safe point to decide whether a commit or abort should be triggered. The main difference between an exposed transaction and a reachable transaction is that: the former, although it has already published its modifications, it can still be aborted (and trigger the cascading abort of other transactions), whereas the latter cannot be aborted anymore. Therefore, it is safe to release all its meta-data without violating the ACO.

Figure 6.4 shows a possible execution using our transactional model. Let γ be the time required for executing the *commit* operation. The following expression shows the total wait time in the case of N transactions.

$$\Sigma_\delta = \alpha + \beta + max(0, N * \gamma - (\alpha + \beta) * \lceil (N - n)/n \rceil)$$

The ord-delay is now limited to the latency of the commit, rather than the time required to make the transaction exposed. Note that a transaction requires less time to commit after being exposed, than to become exposed after executing the try-commit (see Sections 6.5 and 7.1). This implies that the execution time for executing *commit* (γ) should be less, which also reduces the ord-delay. Besides that, threads invest a portion of the stall time executing the operations required to make the transaction exposed, which makes δ negatively proportional with β, unlike other approaches.

Supporting this new model requires handling the following scenarios: *i)* aborted transactions should be able to abort other transactions that accessed their exposed updates; and *ii)* lower age transactions should conflict (and thus abort) with exposed higher age transactions. Accomplishing the above goals requires maintaining some transactional meta-data (e.g., read and write sets, including any acquired locks) even after a transaction is exposed. Those meta-data help in identifying conflicts (or aborting) exposed transactions, and they should be kept accessible until a transaction becomes reachable. Additionally, we need to support

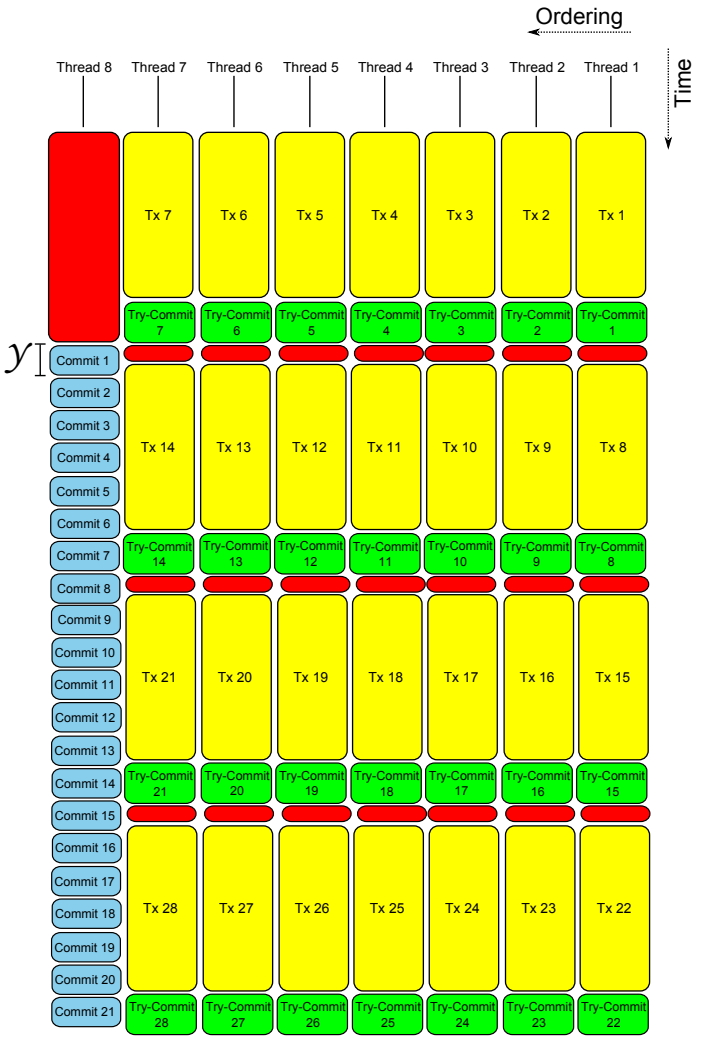


Figure 6.4: The Execution of Ordered Transactions using our approach

the cascading abort of multiple live or exposed transactions that share elements in their read-sets and write-sets. Consequently, transactions must be aware of their interdependencies and construct dependency relations to be able to recover from such situations.

6.5 The Ordered Write Back (OWB)

The Ordered Write Back Algorithm (*OWB*) employs a write-buffer approach; a transaction writes its modifications into a local buffer. While entering the try-commit phase, the transaction acquires a versioned-lock for each object in its write-set and writes its changes to the shared memory, and becomes exposed. To avoid concurrent writers, the locks are not released until the transaction becomes committed or is aborted. However, to allow an early

Operation	Ordering Semantics	*Classical Tx*	*Ordered Tx*	*Conflict Serialization*	*OWB*	*OUL*	*OUL-Steal*
$W_1 \rightarrow R_2$	Speculative Read		✓	✓	✓	✓	✓
$W_1 \rightarrow W_2$	Steal Exclusive Lock		✓	✓			✓
$R_2 \rightarrow W_1$	Invalid Read			✓	✓	✓	✓
$W_2 \rightarrow W_1$	Overwritten Value			✓	✓	✓	✓
$R_1 \rightarrow R_2$		✓	✓	✓	✓	✓	✓
$R_2 \rightarrow R_1$		✓	✓	✓	✓	✓	✓
$R_1 \rightarrow W_2$	Steal Shared Lock		✓	✓	✓	✓	✓
$W_2 \rightarrow R_1$	Invalid Read			✓	✓	✓	✓

Table 6.1: Handling of Read/Write between concurrent transactions.

propagation of the modifications, higher age transactions can access these locked objects. In case an abort is triggered, the exposed transaction is responsible to abort any dependent transaction that has read the exposed values. We use versioning to detect conflicts between concurrent transactions. The transaction performs a validation before exposing its values, and before releasing its locks to approach the final commit.

In practice, for OWB a transaction is exposed if: it is executed until the end without any conflict with other concurrent transactions; it acquired the locks on its modified objects successfully; it exposed their new values to the shared memory; and it is waiting to be reachable. A transaction can commit only if it is reachable and passes the validation of its read operations. The transaction also releases its acquired locks at this stage. As stated earlier, an exposed transaction can still be aborted.

A transaction keeps these meta-data: *1) read-set*, which stores read objects and their read version; *2) write-set*, which stores the modified objects and their new values; and *3) dependencies list*: a list of transactions who read the changes done by this transaction after it becomes exposed. Shared objects are associated with a versioned lock. The lock stores the version number and a reference to the writer transaction (if it exists) that currently owns it. The version is incremented when a new value for the object is exposed.

Algorithms 1, 2 and 3 show the pseudo code of the OWB algorithm. The `Write` operation simply adds the object and its new value to the write-set. The `Read` operation first checks if the object has been earlier modified by the transaction itself. If so, the new value from the write-set is returned; otherwise the object, along with its version, is fetched from the shared memory. If the object is currently exposed, then the writer is aborted only if its age is higher ($W_2 \rightarrow R_1$), and the read operation is retried. If the transaction that holds the lock is older than the reading transaction, we let the latter read the written value ($W_1 \rightarrow R_2$)

Algorithm 1 OWB - Read, Write & Validate

```
1: procedure Begin(Transaction tx)
2:     tx.lastObserved = lastCommitted
3: end procedure

4: procedure Read(SharedObject so, Transaction tx)
5:     readVersion = so.lock.version
6:     currentWriter = so.lock.writer
7:     if tx.writeSet.contains(so) then
8:         tx.readSet.add(so, readVersion)
9:         return tx.writeSet.get(so).value                          ▷ Read written value
10:    else if currentWriter ≠ NULL then                             ▷ Check speculative write
11:        if currentWriter.age > tx.age then
12:            ABORT(currentWriter)                                  ▷ W2 → R1; Read after Speculative Write
13:            go to 5
14:        else                                                      ▷ W1 → R2; Add Tx to its dependencies
15:            currentWriter.dependencies.add(tx)
16:            if currentWriter.status ≠ ACTIVE then                 ▷ Double check writer
17:                ABORT(tx)                                         ▷ Writer got aborted while registration
18:            end if
19:        end if
20:    end if
21:    if readVersion ≠ so.lock.version then
22:        go to 5
23:    end if
24:    Validate_Reads(tx)
25:    tx.readSet.add(so, readVersion)
26:    return so.value
27: end procedure

28: procedure Write(SharedObject so, Object value, Transaction tx)
29:    tx.writeSet.add(so, newValue)                                 ▷ Save new value
30: end procedure

31: procedure Validate_Reads(Transaction tx)
32:    if tx.lastObserved ≠ lastCommitted then
33:        tx.lastObserved = lastCommitted
34:        for each Entry entry in tx.readSet do                     ▷ Validate Read Set
35:            SharedObject so = entry.so
36:            if so.lock.version ≠ entry.readVersion then
37:                return ABORT(tx)                                  ▷ Read a wrong version
38:            end if
39:        end for
40:    end if
41:    return VALID
42: end procedure

43: procedure Validate_Locked_Reads(Transaction tx)
44:    for each Entry entry in tx.readSet do                         ▷ Validate Write Set
45:        SharedObject so = entry.so
46:        if so.lock.writer = tx ∧ so.lock.version ≠ 1 + entry.readVersion then
47:            return ABORT(tx)                                      ▷ Concurrent Expose/Commit Occurs
48:        end if
49:    end for
50:    return VALID
51: end procedure
```

Algorithm 2 OWB - Abort.

```
52: procedure ABORT(TRANSACTION tx)
53:     if tx.status = ABORTED then return false; end if                ▷ Already got aborted
54:     if tx.status = INACTIVE then return false; end if               ▷ Already compeleted
55:     while ! CAS(tx.status, ACTIVE, TRANSIENT) do                    ▷ Try Abort
56:         repeat until tx.status ≠ TRANSIENT                          ▷ Spin Wait
57:         go to 53
58:     end while
59:     for each Transaction dependency in tx.dependencies do
60:         ABORT(dependency)                                           ▷ Abort dependent transactions
61:     end for
62:     for each Entry entry in tx.writeSet do
63:         SharedObject so = entry.so
64:         if so.lock.writer = tx then                                 ▷ Aquired lock
65:             so.value = entry.newValue                               ▷ Revert value
66:             so.lock.writer = NULL                                   ▷ Release the lock
67:         end if
68:     end for
69:     tx.status = ABORTED
70:     return true
71: end procedure
```

and add itself to the writer's dependencies list. That way, if the writer will be aborted in the future, it can cascade its abort to the affected transactions who read its modified objects (see Algorithm 2). It is worth noting that, to avoid inconsistencies while reading from an exposed writer, we let the reader double check the writer state (if it is aborted) after it registers itself in the writer dependencies list; also the dependencies list must provides a thread-safe insertion.

To enter the exposed state, a validation of the read-set is executed to make sure the transaction reads a consistent view of the memory before exposing the locally buffered written objects. `Validate_Reads` (Algorithm 1) compares the current versions of the read objects with the value of the corresponding versions stored in read-set. If the current version is different, then this means that the object was modified after the read, i.e., a Write after Read (WAR) conflict. This modification could be: *speculative*, i.e., the writer is exposed, therefore $R_1 \rightarrow W_2$ or $R_2 \rightarrow W_1$, or *valid*, i.e., the writer committed. In the latter case, the committed writer must have an age older than the current transaction, which means that the current transaction has read an invalid value and it needs to be aborted.

Upon passing a read-set validation, the `Try_Commit` procedure (Algorithm 3) acquires the locks and then writes the write-set to the memory. If the locks are already acquired by another concurrent exposed writer ($W_1 \rightarrow W_2$ or $W_2 \rightarrow W_1$), we handle that by favoring the lower age transaction, and aborting the other. Given that exposed transactions can still be aborted by other transactions, we need to store the old value of modified objects. This is done by swapping the write-set stored values with the old objects' values at commit.

Finally, at `Commit` time we call `Validate_Reads` again to prevent the WAR anomaly. However, given that write-set elements are already locked, we can leverage that to reduce the validation overhead. Consider T_i is executing the commit operation. Let $X \in$ read-set$(T_i) \cap$ write-set(T_i). As T_i is still acquiring the locks over its write-set (including X), T_i is sure that

Algorithm 3 OWB - Commit.

```
72: procedure TryCommit(Transaction tx)
73:     if tx.status = ABORTED then return false; end if                    ▷ Already got aborted
74:     while ! CAS(tx.status, ACTIVE, TRANSIENT) do                        ▷ Try Commit
75:         repeat until tx.status ≠ TRANSIENT                              ▷ Spin Wait
76:         return false
77:     end while
78:     if VALIDATE_READS(tx) ≠ VALID then return false; end if
79:     for each Entry entry in tx.writeSet do                              ▷ Lock Write Set
80:         SharedObject so = entry.so
81:         currentWriter = so.lock.writer
82:         if currentWriter ≠ NULL then
83:             if tx.age < currentWriter.age then
84:                 ABORT(currentWriter)                                    ▷ W2 → W1; Write after Specu. Write
85:             else
86:                 ABORT(tx)                                               ▷ W1 → W2; Write after Write
87:                 return false
88:             end if
89:             if ! CAS(so.lock.writer, NULL, tx) then                     ▷ Acquire Lock
90:                 go to 81
91:             end if
92:         end if
93:     end for
94:     for each Entry entry in tx.writeSet do
95:         SharedObject so = entry.so
96:         so.lock.version + +
97:         temp = so.value                                                 ▷ Save old value
98:         so.value = entry.newValue                                       ▷ Expose written value
99:         entry.newValue = temp
100:    end for
101:    if Validate_Locked_Reads(tx) ≠ VALID then return false; end if
102:    tx.status = ACTIVE                                                  ▷ Transaction Exposed
103:    return true
104: end procedure

105: procedure Commit(Transaction tx)
106:    if tx.status = ABORTED then return false; end if                    ▷ Already got aborted
107:    while ! CAS(tx.status, ACTIVE, TRANSIENT) do                        ▷ Try Complete
108:        repeat until tx.status ≠ TRANSIENT                              ▷ Spin Wait
109:        return false
110:    end while
111:    if VALIDATE_READS(tx) ≠ VALID then return false; end if
112:    for each Entry entry in tx.writeSet do
113:        SharedObject so = entry.so
114:        so.lock.writer = NULL                                           ▷ Unlock
115:    end for
116:    tx.status = INACTIVE                                                ▷ Transaction Committed
117:    return true
118: end procedure
```

the value of X is unchanged since its lock acquisition, thereby it could be excluded from the commit-time read-set validation. To do so, it requires checking that read-set objects have not been changed while acquiring locks, thus `Validate_Locked_Reads` is called after that.

Keeping the commit execution time short is fundamental as it impacts the ord-delay (as shown in Section 6.4.1); therefore, the optimization just described shrinks the commit execution at the price of adding an extra check in the `Try-Commit` procedure (i.e., `Validate_Locked_Reads` call). However, having an object read and written in a transaction is a common pattern, which makes this optimization fruitful. In addition, if there is no concurrent transaction that committed since the beginning of the execution, invoking `Validate_Reads` and `Validate_Locked_Reads` can be avoided (For clarity, this optimization is not included in the pseudo code). To do so, the transaction stores the age of the last committed transaction at its beginning, and checks if it is changed before the validation. Finally, when a transaction becomes reachable and the re-validation succeeds, the `commit` operation releases its acquired locks and reclaims any transactional meta-data.

As correctness guarantee, OWB guarantees TMS1 [62]. The intuition is that: if for a the history generated by OWB, every exposed transaction is committed, then the history is opaque [76]. First of all, transactions can commit only in the ACO order, serializing all the committed transactions, making OWB strict serializable. Moreover, OWB allows transactions to read only from commit-pending (exposed) and committed transactions, and any time a transaction enters the exposed state, it aborts all concurrent transactions that has read a value that violates the ACO. However, exposed transactions can abort after some live transaction already read those values. This is not allowed by Opacity, but TMS1 allows that as long as the live transactions do not perform any operation after the exposed transaction is aborted. OWB implements that through an atomic cascading abort. We give more details about correctness in Section 6.7.

6.6 Implementation

6.6.1 Lock Structure

In this section, we detail the implementation of the locks. Each memory address is associated with a 32-bit lock. The mapping between addresses and locks is made by leveraging the least significant bits, so a single lock is able to cover multiple addresses. The lock is divided into two sections: the most significant N bits represent the reference to the writer, and the remaining bits represent the version number.

Algorithm 4 Thread Execution

```
1: for each Transaction tx in WorkQueue do
2:     if validator = IDLE ∧ CAS(validator, IDLE, BUSY) then
                                                          ▷ Try to be the validator
3:         tx = ExposedList[last_committed]
4:         if tx = NULL then                              ▷ Tx is not exposed yet
5:             go to 16                                   ▷ Stop validation
6:         end if
7:         if tx.commit() = FAIL then                     ▷ Perform Tx commit
8:             tx.start()
9:             tx.execute()                               ▷ Reexecute failed transaction
10:            tx.tryCommit()                             ▷ Commit without validation
11:            tx.commit()
12:        end if
13:        last_committed++
14:        CommittedQueue.enqueue(tx)
15:        go to 3                                        ▷ Validate next exposed Tx
16:        validator = IDLE                               ▷ Release the validator role
17:    end if
18:    if aborts>LIMIT ∨ tx.age - last_committed > MAX then
19:        while tx.age - last_completed > MIN do
20:            for each Transaction tx in CommittedQueue do
21:                tx.clean()                             ▷ Do housekeeping
22:            end for
23:        end while
24:    end if
25:    tx.start()
26:    tx.execute()                                       ▷ Execute transaction
27:    if tx.tryCommit() = FAIL then                      ▷ Try to expose transaction
28:        go to 25                                       ▷ Retry
29:    end if
30:    ExposedList[tx.age] = tx                           ▷ Add to pending transactions
31: end for
```

6.6.2 Thread Execution

In our implementation, a single executing thread plays multiple roles: *worker*, *validator*, or *cleaner*. Switching between different roles is done according to the procedure in Algorithm 4. A *worker* thread executes the transaction (creation, reads, and writes) and performs the `try-commit`; while a thread in the *cleaner* role takes care of the housekeeping (reclaiming transactional meta-data). There is a single thread at a time in the *validator* role, which is responsible for moving commit-pending (exposed) transactions to the committed state and also re-executes invalid transactions. We adopt the *flat combining* [87] technique for enabling threads to take the *validator* role as shown.

In OWB, exposed transactions acquire locks on their write-set, which limits concurrency and increases the conflict probability. For this reason, it is highly desirable for a transaction to stay a short time in the state between exposed and committed. Therefore, the worker threads should avoid overwhelming the validator by executing (and exposing) many transactions. In line 18, if the count of pending exposed transactions exceeds a threshold or the number of aborts increases, a worker thread spends its cycles cleaning the committed transactions (i.e., it switches its role from worker to cleaner). It continues cleaning until a minimum threshold

for the count of those pending transactions is reached (line 19).

6.7 Correctness

In this section, we discuss the correctness guarantees of the given algorithms.

First, we show how OWB preserve the ACO. Suppose by contradiction that the ACO is violated. Let T_i and T_j be two transactions such that $T_i \prec T_j$. The interesting case is if T_i successfully reads a value of an object X written by T_j. This implies that $R_i(X)$ happened after T_j exposes X's value. In OWL, T_i acquires a shared lock on X at the time of the read operation, by checking if there is no writer. In order to have a successful read, the shared lock must be acquired, thus the write lock should not be already granted. This implies that T_j has released all its locks. As a transaction does not release its acquired locks until it *commits*, T_j must be necessarily committed. Therefore $R_i(X)$ must occur after $commit(T_j)$. Since a transaction cannot perform any step after it commits, $R_i(X) \rightarrow commit(T_i)$. This means $commit(T_j) \rightarrow commit(T_i)$, which cannot be the case since they must commit in order, according to their ages.

Now we prove that OWB is serializable. In order to prove that, we define $DG(i, j)$ as a predicate defining a dependency between T_i and T_j, when T_j reads a value written by T_i, or T_j overwrites a value written by T_i. Using this definition, we can construct a dependency directed graph $DG(T, D)$, where T is the set of all committed transactions, and D is the set of dependency relations. It is easy to see that $DG \subset SG$, where SG is the conflict serialization graph [21]. A history is serializable if and only if its SG is acyclic. Note that serializability is not guaranteed if DG is acyclic. Assume by contradiction that an execution of our algorithms produce a cyclic DG, which implies having an edge $D(i, j)$ where $i > j$. By definition of dependency, this means that either T_j reads a value written by T_i (i.e., $W_i(X) \rightarrow R_j(X)$), or T_j overwrites a T_i's written value (i.e., $W_i(X) \rightarrow W_j(X)$). In all the proposed algorithms, exclusive locks must be acquired when we expose the written values (at commit time) and released only at commit, or passed to a higher age transaction (which is not the case here). We can rewrite the previous situations as $commit(T_i) \rightarrow R_j(X)$ or $commit(T_i) \rightarrow W_j(X)$. Since a transaction cannot perform any step after it commits, $commit(T_i) \rightarrow commit(T_j)$, which cannot be the case as mentioned earlier; therefore, DG is a acyclic. Assume $e \in E = SG \setminus DG$, this edge represents the case where $R_i(X) \rightarrow W_j(X)$, which means $R_i(X) \rightarrow commit(T_j)$. In OWB, the procedure `Validate_Reads` captures this by comparing the read version with the current version of the accessed object (Algorithm 1). So $E = \emptyset \Rightarrow SG = DG \Rightarrow SG$ is acyclic, making the algorithms serializable.

Note that the serialization point for OWB is line 116 in Algorithm 1. As the serialization point is inside the transaction execution, all the algorithms preserve the real-time order, and are strict serializable.

In addition to being strict serializable, OWB is also TMS1 [62], a stronger condition than

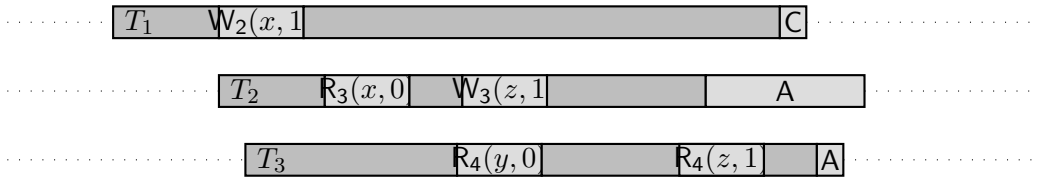


Figure 6.5: An execution of OWB, which is TMS1. Initial value of all the shared variables is 0. The ACO is $T_1 \prec T_2 \prec T_3$

strict serializability. This means that every history produced by OWB is TMS1. Being TMS1, OWB ensures that response of every object operation, even by aborted and live transactions, is consistent with a serial execution. Informally, for a history to be TMS1, it must be strict serializable, and for every successful response of an object operation by a transaction T, there must exist a serialization of a subset of the transactions, justifying the response. This subset must contain T (till the response) and all the committed transactions that completed before T started. In addition to them, the serialization can also contain some commit-pending transactions, and some committed and even aborted transactions, that are concurrent to T. We have already shown earlier that OWB is strict serializable. Since OWB allows a read operation to return a value written by an exposed transaction, which may get aborted later, it justifies including concurrent aborted transaction for the response. Recall that OWB allows reading values written by committed and exposed transactions only, but not from aborted transactions. The intuition is that if a transaction reads from an exposed transaction, which gets aborted later, the reading transaction is also aborted without executing any further operations. This is done using cascading mechanism in OWB (line 60 in Algorithm 1). Figure 6.5 shows an execution of OWB. In this execution, the transaction T_3 reads the value of z written by an exposed transaction (T_2), which gets aborted later due to the commit of T_1 that writes a new value of x.

6.8 Evaluation

We implemented OWB, the ordered version of four existing well-known TM designs (i.e., TL2 [59], NOrec [53], and UndoLog [69] with and without visible readers) and STMLite [120], and compared their performance using a set of micro-benchmarks and STAMP [37], where an order has been enforced. The evaluation of OWB is postponed to Section 7.4 after introducing the OUL algorithm in the next chapter.

Chapter 7

Ordered Undolog Algorithm

The Ordered Undo Log (*OUL*) Algorithm is an undo-log algorithm that preserves the ACO. Here, transactional updates affect the shared memory at encounter time, while the old value is kept in a local undo-log. Such a scheme implies that the transactions' order is guaranteed while operations are invoked, and not at commit time as in OWB. In order to deploy the above idea, each object is associated with a read-write lock. The transaction acquires a read or write lock according to its need, as explained later. Also, each lock stores the reference to the (single) writer transaction, which can be either the current transaction holding the lock or the one that committed that version, and a list of concurrent readers, namely those transactions that accessed the version for reading it, and they are still live or commit-pending. Note that the size of the readers list impacts the efficiency of the protocol, thus it should be bounded.

As in OWB, every transaction in OUL maintains a write-set, but here the write-set stores the old values of the written objects (undo-log). Regarding the transaction read-set, it is implicitly represented by the object lock's readers list.

7.1 Ordered Undolog Algorithm (OUL)

In *OUL*, we use an immediate update approach with undo logging for recovery from aborts. Accessed objects are locked at encounter time using a variant of read-write locks. Our lock permits a single writer to co-exists with multiple readers under certain conditions. The lock keeps a bounded list of the reader transactions, *visible readers*, and a reference to the writer transaction, hence, dependency between transactions is preserved through the read-write locks.

Algorithms 5 and 6 show the OUL core operations[1]. In the `Read`, we allow Read after Write

[1]We omitted the necessary FENCE instructions to prevent out-of-order execution. Those can be found

(RAW) conflicts only if the writer transaction has a lower age ($W_1 \rightarrow R_2$); otherwise the speculative writer is aborted ($W_2 \rightarrow R_1$). The `Write` operation enforces that only a single transaction can hold the write lock on the object at a time. A Write-Write conflict is solved by aborting the transaction with the lowest age. As readers are visible, the writer transaction can check if there is any (wrong) speculative reader, and abort it accordingly ($R_2 \rightarrow W_1$).

Algorithm 5 OUL - Read & Write

```
1: procedure READ(SHAREDOBJECT so, TRANSACTION tx)
2:     if tx.status = TRANSIENT then return ABORT(tx) end if
3:     Transaction currentWriter = so.lock.writer;
4:     if currentWriter = BUSY  then go to 9 end if
5:     if currentWriter ≠ NULL ∧ currentWriter.status ≠ INACTIVE ∧ currentWriter.age > this.age then
6:         ABORT(currentWriter)                                   ▷ W2 → R1; Read after specu. Write
7:         go to 9
8:     end if
9:     registered = false
10:    repeat
11:        for i=0 to MAX_READERS do
12:            Transaction readerSlot = so.lock.reader[i]
13:            if readerSlot ≠ ACTIVE ∧ readerSlot ≠ PENDING ∧ CAS(so.lock.reader[i], readerSlot, tx) then
14:                registered = true                                ▷ Found empty reader slot
15:            end if
16:        end for
17:    until registered
18:    if currentWriter ≠ so.lock.writer then                     ▷ Writer was changed
19:        go to 9
20:    end if
21:    return so.value
22: end procedure

23: procedure WRITE(SHAREDOBJECT so, OBJECT value, TRANSACTION tx)
24:    if tx.status = TRANSIENT then ABORT(tx); end if
25:    Transaction currentWriter = so.lock.writer;
26:    if currentWriter = BUSY  then go to 24 end if
27:    if currentWriter ≠ tx  then                                ▷ Already in write-set
28:        if currentWriter ≠ NULL ∧ currentWriter.status ≠ INACTIVE then
29:            if currentWriter.age > this.age  then
30:                ABORT(currentWriter)                           ▷ W2 → W1; Write after specu. Write
31:                go to 24
32:            else
33:                ABORT(tx)                                      ▷ W1 → W2; Write after Write
34:            end if
35:        end if
36:        if  ! CAS(so.lock.writer, currentWriter, BUSY)  then
37:            go to 24                                           ▷ Failed to aquire the lock
38:        end if
39:        tx.writeSet.add(so, so.value)                          ▷ Save old value
40:    end if
41:    for i=0 to MAX_READERS do
42:        Transaction readerSlot = so.lock.reader[i]
43:        if readerSlot ≠ INACTIVE ∧ readerSlot.age > tx.age) then
44:            ABORT(readerSlot)                                  ▷ R2 → W1; Abort specu. reader
45:        end if
46:    end for
47:    so.value = newValue                                        ▷ Write new value
48:    so.lock.writer = tx                                        ▷ Save me as the new writer
49: end procedure
```

in the source code at: `https://bitbucket.org/mohamed-m-saad/ordertm`.

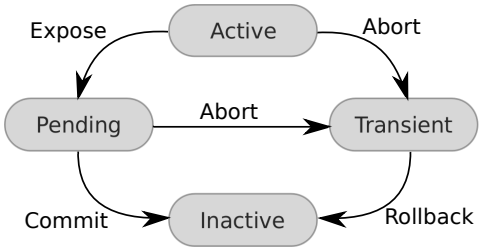


Figure 7.1: OUL Transaction States

One of the major benefit of a write through protocol is that the `Try-Commit` procedure is simple because the values are already in the shared memory. However, in OUL exposing a transaction only means that it did not conflict with other transactions so far – but it could be still aborted to preserve the ACO. In the `Commit` procedure, the transaction is marked as `Inactive` and locks are released. As we said before, given that a lock is maintained with a back-reference to the transaction that holds it, setting the transaction status is sufficient to release all the locks held by that transaction with a single step. On the other hand, in `Abort` the transaction has to restore old values from the undo-log (Rollback), and release all the locks (switches to the `Inactive` state).

Algorithm 6 OUL - Commit & Abort

```
50: procedure TryCommit(Transaction tx)
51:     if !CAS(tx.status, ACTIVE, PENDING) then ABORT(tx) end if
52: end procedure

53: procedure Commit(Transaction tx)
54:     if CAS(tx.status, PENDING, INACTIVE) then return true end if
55:     repeat until tx.status ≠ TRANSIENT
                                                          ▷ Wait till be aborted
56: end procedure

57: procedure Abort(Transaction tx)
58:     if tx.status = INACTIVE then return true; end if
                                                          ▷ Check if already aborted
59:     if CAS(tx.status, PENDING, TRANSIENT) then
                                                          ▷ Rollback
60:         for each Entry entry in tx.writeSet do
61:             SharedObject so = entry.so
62:             Object value = entry.value
63:             so.value = value                          ▷ Restore old value
64:             for i=0 to MAX_READERS do
65:                 Transaction readerSlot = so.lock.reader[i]
66:                 if readerSlot ≠ INACTIVE ∧ readerSlot.age > tx.age) then
67:                     ABORT(readerSlot)                 ▷ Abort specu. reader
68:                 end if
69:             end for
70:         end for
71:         tx.status = INACTIVE
72:     else                                              ▷ Set aborted
73:         return CAS(tx.status, ACTIVE, TRANSIENT)
74:     end if
75: end procedure
```

Figure 7.1 recaps the different states of OUL transactions, and their transitions. The `Active`, `Pending` and `Inactive` states correspond to the live, exposed, and committed (or aborted) situations respectively, while the `Transient` state indicates the *to-be-aborted* event, which could be triggered by another conflicting transaction.

7.1.1 The OUL-Steal Algorithm

In this section, we introduce OUL-Steal, a variant of the OUL algorithm where we relax the aforementioned multiple-writers restriction and allow write-writer conflicts while guaranteeing ACO. In both OWB and OUL, conflicting transactions co-operate to commit as they are allowed to proceed without aborts even in the presence of some read-write conflict, as long as ACO is still preserved. However, a writer transaction holds the locks until reaching the *commit* state, which sometimes limits the overall concurrency.

Let T_i and T_j be two conflicting writers on an object X, and $T_i \prec T_j$. In OUL, if T_i finds X already locked by T_j ($W_1 \rightarrow W_2$), T_i should abort T_j. However, ACO could be still preserved if T_j overwrites the value of T_i, as long as there is no other transaction T_k, with $T_i \prec T_k \prec T_j$, that will read X in the future.

OUL-Steal allows a transaction with higher age to overwrite the value written by a concurrent transaction with lower age ($W_1 \rightarrow W_2$), and *steal* its lock. The newer transaction stores the stolen lock in a local list so that it can be returned back to the original writer (the transaction with the lower age) in case of abort. That way, if a mid-age reader T_k needs the value of an older age transaction, then it can abort the newer transaction(s) which stole the lock(s); otherwise (i.e., without T_k), the value written by the newer age transaction will be used by the higher age readers. This operation could be repeated at multiple levels until the reader reaches the correct writer transaction. More formally, $\forall i > k$, if $X \in$ read-set(T_k) and $T_i \in$ writers(X), then T_k aborts T_i.

In `Write`, the lock is passed to the higher age writer and is saved in its write-set. As a consequence, the written address exists in the undo-log of both the writers (the original and the one which stole the lock). During the `Abort`, the transaction uses `Rollback` to revert its changes using its undo-log. An undo-log entry can be:

- *stolen* by another writer: which means the transaction does not have the ownership record at the abort time. In this case, the transaction does not do any action, although, it keeps the undo-log entry, which contains the address value before the current transaction modifications.
- *exclusively modified* by the current transaction, reverting the old value from the undo-log, and aborting the speculative readers.
- *stolen from* another writer: in addition to the steps done in the *exclusively modified* case, the lock ownership is passed back to the old writer, and the current transaction

checks the state of the old writer. If it was *not aborted*, then no further action is needed. Otherwise, the transaction calls the `Rollback` of the old owner. At this stage, the old writer will treat the entry as the cases of *exclusively modified* or *stolen from*, accordingly.

The complete pseudo code of the OUL-Steal algorithm is shown in Algorithms 7 and 8.

Algorithm 8 OUL-Steal - Abort

```
58: procedure ROLLBACK(TRANSACTION tx)
59:     tx.aborted = true
60:     for each Entry entry in tx.writeSet do
61:         SharedObject so = entry.so
62:         if CAS(so.lock.writer, tx, BUSY) then
63:             Object value = entry.value
64:             so.value = value                                          ▷ Restore old value
65:             so.lock.writer = entry.originalOwner                      ▷ Release lock, or return the original owner
66:             if entry.originalOwner != NULL ∧ entry.originalOwner.aborted then
67:                 ROLLBACK(entry.originalOwner)
68:             end if
69:         end if
70:         for i=0 to MAX_READERS do
71:             Transaction readerSlot = so.lock.reader[i]
72:             if readerSlot ≠ INACTIVE ∧ readerSlot.age > tx.age) then
73:                 ABORT(readerSlot)                                     ▷ Abort speculative readers
74:             end if
75:         end for
76:     end for
77: end procedure

78: procedure ABORT(TRANSACTION tx)
79:     if tx.status = INACTIVE then return true; end if                  ▷ Already Aborted
80:     if CAS(tx.status, PENDING, TRANSIENT) then
81:         Rollback(tx)                                                  ▷ Rollback
82:         tx.status = INACTIVE
83:     else
84:         if CAS(tx.status, ACTIVE, TRANSIENT) then                     ▷ Set aborted
85:             return true
86:         end if
87:     end if
88:     return false                                                      ▷ Failed to abort
89: end procedure
```

In `Write`, the lock is passed to the higher age writer and is saved in its write-set (Lines 34 and 40). The `Abort` procedure checks that the current transaction is still holding the lock. If so, in addition to reverting the old value or the written objects and aborting the speculative readers, the lock is passed back to the old owner (if exists). If the lock has been stolen by some other transaction, that transaction is now in charge of restoring the value by calling `Rollback` recursively over the chain of previously aborted owner(s) transaction(s) (if exists) (Line 67). However, if the original owner of the lock is still alive (or committed), then it is sufficient for the current owner to just revert the newly written value of the object from the undo-log.

Algorithm 7 OUL-Steal - pseudo code

```
1: procedure Read(SharedObject so, Transaction tx)
2:     if tx.status = TRANSIENT then return ABORT(tx) end if
3:     Transaction currentWriter = so.lock.writer;
4:     if currentWriter = BUSY  then go to 2 end if
5:     if currentWriter ≠ NULL ∧ currentWriter.status ≠ INACTIVE ∧ currentWriter.age > this.age then
6:         ABORT(currentWriter)                                   ▷ W2 → R1; Read after Speculative Write
7:         go to 2
8:     end if
9:     registered = false
10:    repeat
11:        for i=0 to MAX_READERS do
12:            Transaction readerSlot = so.lock.reader[i]
13:            if readerSlot ≠ ACTIVE ∧ readerSlot ≠ PENDING ∧ CAS(so.lock.reader[i], readerSlot, tx) then
14:                registered = true                                          ▷ Found empty reader slot
15:            end if
16:        end for
17:    until registered
18:    if currentWriter ≠ so.lock.writer then                     ▷ Writer got changed meanwhile
19:        go to 2
20:    end if
21:    return so.value
22: end procedure

23: procedure Write(SharedObject so, Object value, Transaction tx)
24:    if tx.status = TRANSIENT then ABORT(tx); end if
25:    Transaction currentWriter = so.lock.writer
26:    if currentWriter = BUSY  then go to 24 end if
27:    if currentWriter ≠ tx  then                                ▷ Already in write-set
28:        steal = false
29:        if currentWriter ≠ NULL ∧ currentWriter.status ≠ INACTIVE then
30:            if currentWriter.age > this.age  then
31:                ABORT(currentWriter)                           ▷ W2 → W1; Write after Specu. Write
32:                go to 24
33:            else
34:                steal = true                                   ▷ W1 → W2; Lock Steal, Write after Write
35:            end if
36:        end if
37:        if  ! CAS(so.lock.writer, currentWriter, BUSY)  then   ▷ Aquire the lock
38:            go to 24
39:        end if
40:        tx.writeSet.add(so, so.value, steal ? currentWriter : NULL) ▷ Save old value, and old writer when stealing the lock
41:    end if
42:    for i=0 to MAX_READERS do
43:        Transaction readerSlot = so.lock.reader[i]
44:        if readerSlot ≠ INACTIVE ∧ readerSlot.age > tx.age) then
45:            ABORT(readerSlot)                                  ▷ R2 → W1; Abort speculative readers
46:        end if
47:    end for
48:    so.value = newValue                                        ▷ Write new value
49:    so.lock.writer = tx                                        ▷ Save me as the new writer
50: end procedure

51: procedure TryCommit(Transaction tx)
52:    if  !CAS(tx.status, ACTIVE, PENDING) then ABORT(tx) end if
53: end procedure

54: procedure Commit(Transaction tx)
55:    if  CAS(tx.status, PENDING, INACTIVE) then return true end if
56:    repeat until tx.status ≠ TRANSIENT                         ▷ Wait till be aborted
57: end procedure
```

7.2 Implementation

7.2.1 Lock Structure

Each memory address is associated with a 32-bit lock. The mapping between addresses and locks is made by leveraging the least significant bits, so a single lock is able to cover multiple addresses. The lock is divided into two sections: the most significant N bits represent the reference to the writer, and the remaining bits represent the header address of the readers list. We use a bounded list of readers to limit the number of concurrent readers, which is set to 10 transactions in our experiments.

7.2.2 Thread Execution

Threads executes similarly to the mechanism discussed in Section 6.6.2. However, the situation is different in OUL (and its variant), as it uses encounter time locking, thus the duration of the period where the transaction is in the state between exposed and committed does not affect the performance. Nevertheless, moving the transaction to the committed state releases the locks earlier, hence reducing the conflict probability.

7.3 Correctness

As correctness guarantee, OUL guarantees Strict Serializability [139]. Unlike OWB, OUL allows reading from live transactions, which is not allowed by TMS1 (and hence opacity). However, similar to OWB, OUL restricts transactions to commit only in the ACO order, making OUL strict serializable.

In order to show how OUL and OUL-Steal preserve the ACO, we will borrow the same idea we used in Section 6.7. Suppose by contradiction that the ACO is violated. Let T_i and T_j be two transactions such that $T_i \prec T_j$. The interesting case is if T_i successfully reads a value of an object X written by T_j. This implies that $R_i(X)$ happened after $write(T_j)$ in OUL. In both OUL and OUL-Steal, T_i acquires a shared lock on X at the time of the read operation using visible reads. In order to have a successful read, the shared lock must be acquired, thus the write lock should not be already granted. This implies that T_j has released all its locks. As a transaction does not release its acquired locks until it *commits*, T_j must be necessarily committed. Therefore $R_i(X)$ must occur after $commit(T_j)$. Since a transaction cannot perform any step after it commits, $R_i(X) \rightarrow commit(T_i)$. This means $commit(T_j) \rightarrow commit(T_i)$, which cannot be the case since they must commit in order, according to their ages.

The serializablility guarantee for both OUL and OUL-Steal is proved using the same tech-

nique used in Section 6.7 (i.e., using the conflict serialization graph). Assume $e \in E = SG \setminus DG$, this edge represents the case where $R_i(X) \rightarrow W_j(X)$, which means $R_i(X) \rightarrow commit(T_j)$. In both OUL and OUL-Steal, the readers' visibility enables T_j to detect the $R_i(X)$ and aborts it (line 43 in Algorithm 5). So $E = \emptyset \Rightarrow SG = DG \Rightarrow SG$ is acyclic, making the algorithms serializable.

Note that the serialization point for OUL is line 54 in Algorithm 6. As the serialization point is inside the transaction execution, all the algorithms preserve the real-time order, and are strict serializable.

Opacity [76] is an important correctness property for TM implementations however, given the fundamental characteristic of our proposals of letting threads cooperate before transactions are actually completed, we cannot claim to ensure opacity. More specifically, the two OUL algorithms forward a written value encounter time, thus they allow doomed transaction to observe inconsistent state.

7.4 Evaluation

We compare our algorithms with STMLite [120]: a light-weight STM with ACO support used to support code parallelization, and an ordered version of three state-of-art TM algorithms: TL2 [59], NOrec [53] and UndoLog [69] (with and without visible readers). Both TL2 and NOrec follow the write-back design strategy and validate transactions at commit time. Given that, ordering commit operations according to ages guarantees that transactions see a consistent view of the memory before modifying the shared state. In order to aid the ordering for UndoLog, we exploit an age-based contention policy (i.e., always favor transactions with the lower age) to handle write-write conflicts. In the *visible readers* variant, the writer transaction aborts all active readers, while when readers are *invisible* the writer retries multiple times if the object is locked, then it backs off.

STMLite uses a write-back implementation and replaces the need for constructing a read-set by leveraging signatures (Bloom Filters). There is a tradeoff in determining the effective size of signatures, but the authors recommended a range of 32 to 1024. In our experiments, we used a signature of size 64 with the STL hashing function because it provided the best performance. The number of threads in STMlite also includes its commit manager.

All competitors, including STMLite, have been re-implemented using the same baseline software framework, so that all take advantage of the same low-level optimizations.

We conducted the experiments on an AMD machine equipped with 2 Opteron 6168 CPUs, each with 12-core running at 1.9 GHz. The total memory available is 12 GB and the cache sizes are 128 KB for the L1, 512 KB for the L2, and 12 MB for the L3. Results are the average of five runs. We report the throughput for micro benchmarks and the application execution time for STAMP by varying the number of threads used (the datapoint at 1 thread

shows the performance of the single-threaded transactional execution). For completeness, the performance of the non-transactional single-threaded execution (green line) has also been included. In all the benchmarks, experiments are conducted by parallelizing the main for-loop that activates all transactions. The ACO is defined according to the index of the loop. In other words, the first iteration of the loop activates the transaction that must commit first, and so on. Note that this technique is the standard way to parallelize sequential code (e.g., [120]). Also, it does not introduce any additional contention on the benchmark itself.

7.4.1 Micro Benchmark

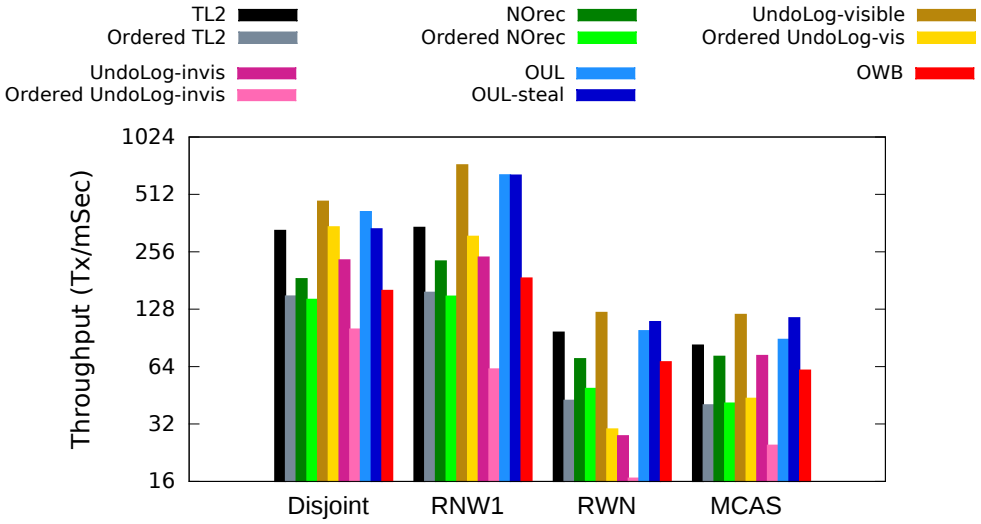
In our first set of experiments we consider the RSTM micro-benchmarks [2] to evaluate the effect of different workload characteristics, such as the amount of operations per transaction, the transaction length, and the read/write ratio, on the performance. Each experiment included running half million transactions. For all micro benchmarks, we configured three types of transactions: short, long, and heavy. Both *short* and *heavy* have the same number of accesses, but the latter adds more local computation in between them. Such a workload is representative of workloads produced by parallelization frameworks. *Long* transactions simply produce more transactional accesses.

Figure 7.2 summarizes the peak performance of all competitors. From that we can see the gap in performance between the ordered and unordered versions of the same algorithm: 26-56% for TL2, 13-41% for NOrec, 12-88% for UL-vis, and 28-74% for UL-invis.

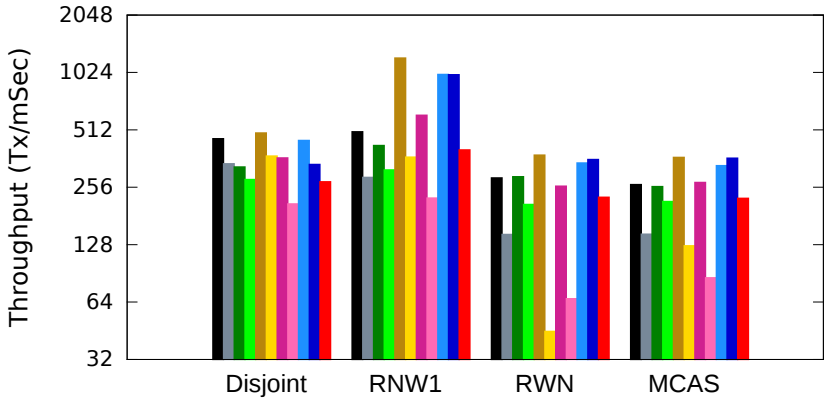
As a general comment on the results, OUL and OUL-Steal outperform all other ordered versions of the algorithms. OUL-Steal excels for write loads and performs equally to OUL in read loads; OWB outperforms all write-back based implementations in most benchmarks. At high thread count, STMLite suffers from false conflicts due to the use of signatures. However, at low number of threads (less than 8) and with *Long* transactions it achieves a higher peak throughput than Ordered TL2 and Ordered NOrec, because it benefits from the quick validation using signatures. For the UL-inv algorithm, we found that the readers' visibility was crucial; without this information, the algorithm may abort a lower age transaction (using timeout) while some higher age transaction holds the read shared lock. On the other hand, these higher age transactions cannot commit before their order comes, hence they timeout.

In configurations where the performance of the sequential (non-transactional) execution is faster than many ordered algorithms, our solutions outperform it, letting parallelism pay off. However, there are two benchmarks configured with long transactions where the sequential execution is faster. These workloads are not suited to be parallelized while providing a commit order because of the high abort cost, which is more impacting given the chance to encounter context switches, threading overhead, and cache thrashing.

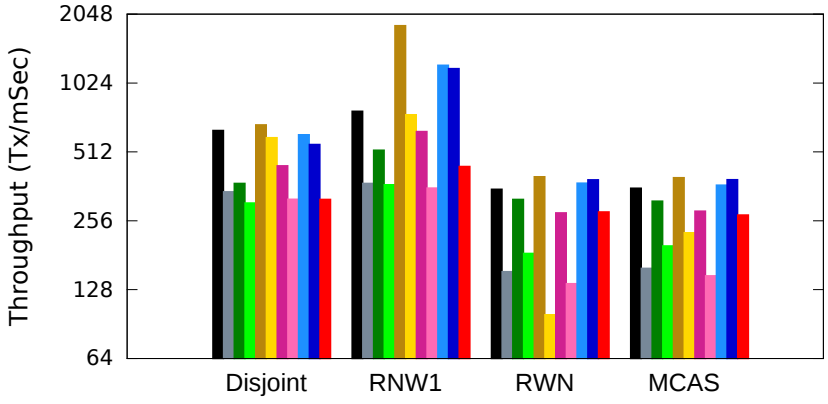
The *DisjointBench* (Figures 7.3a-7.3c) produces a workload with no conflict between concurrent transactions. Every transaction accesses a different set of addresses with read and write



(a) Long Transaction

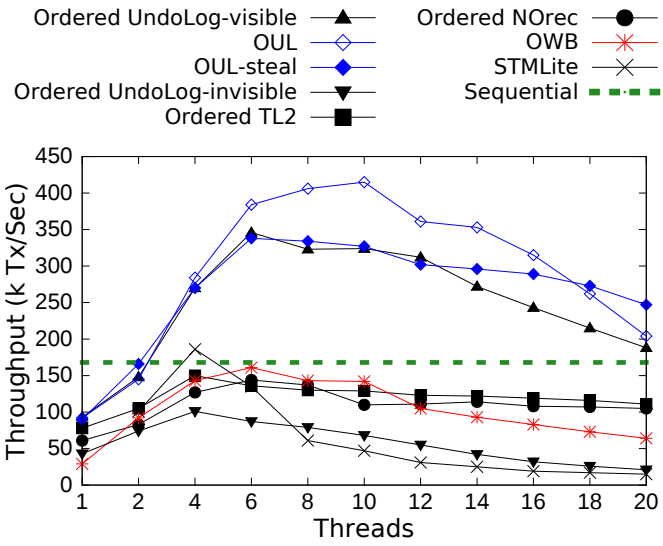


(b) Short Transaction

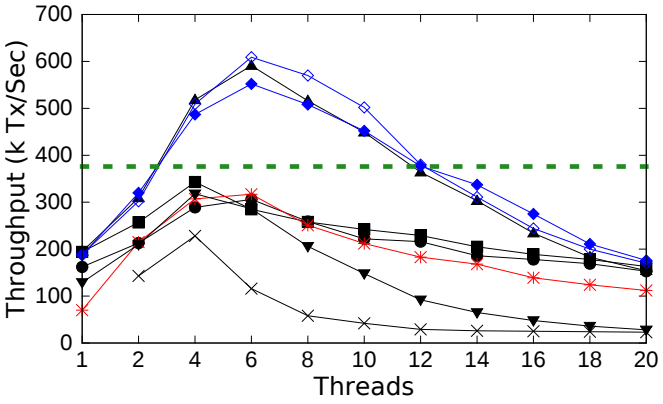


(c) Heavy Transaction

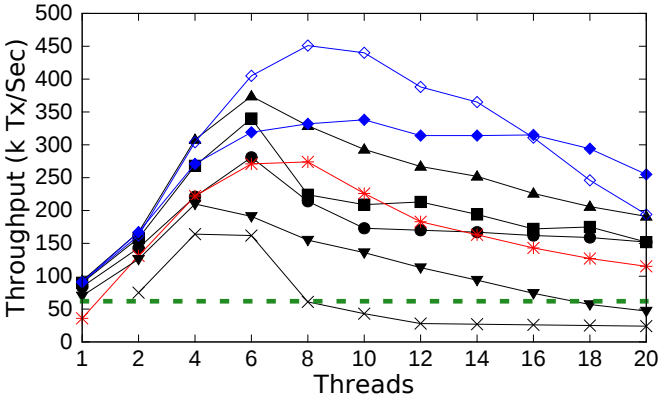
Figure 7.2: Peak performance of all competitors (including unorderd) using all micro benchmarks (Y-axis is log scale).



(a) Disjoint-Long

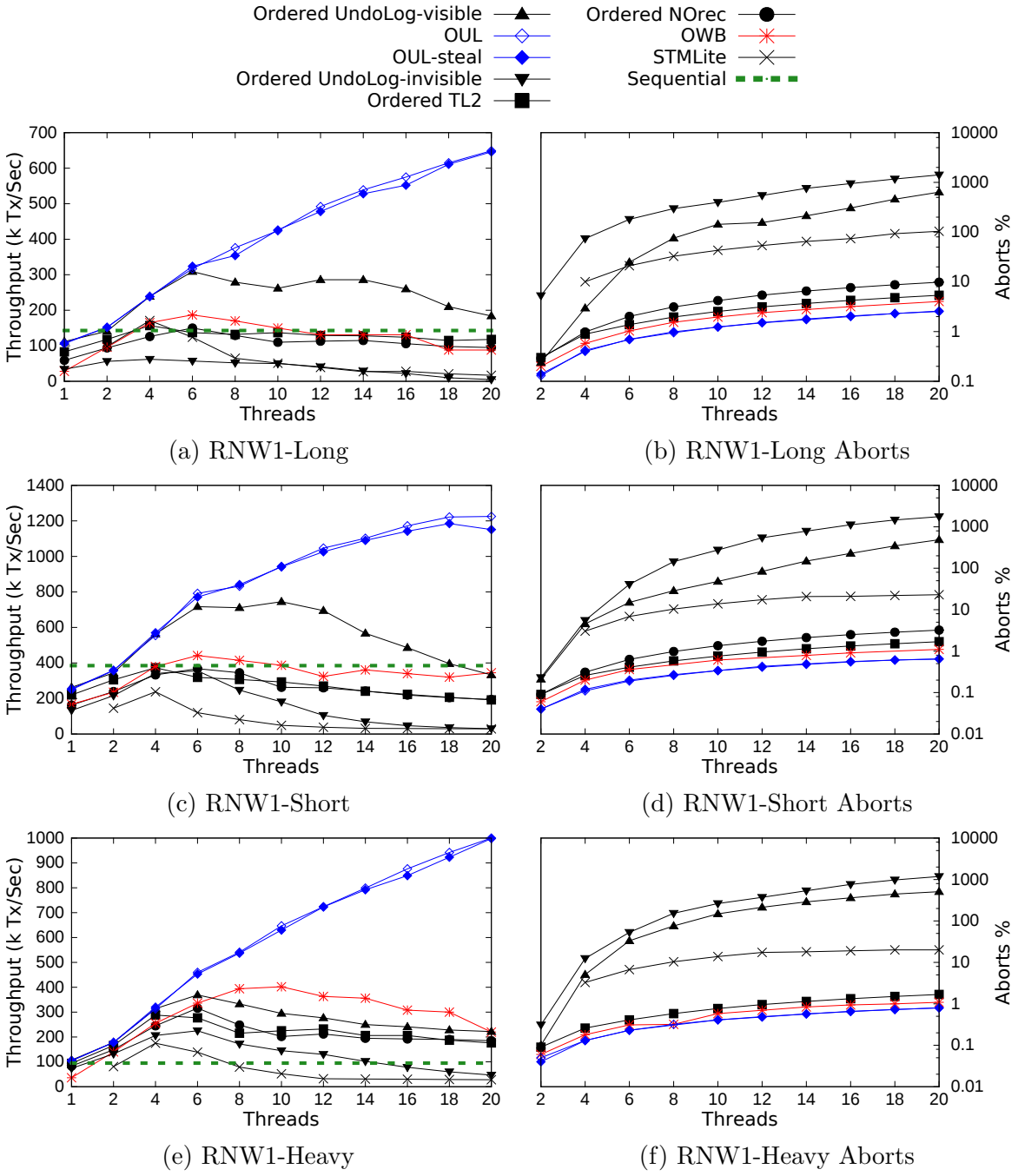


(b) Disjoint-Short



(c) Disjoint-Heavy

Figure 7.3: Disjoint Benchmark.



(a) RNW1-Long

(b) RNW1-Long Aborts

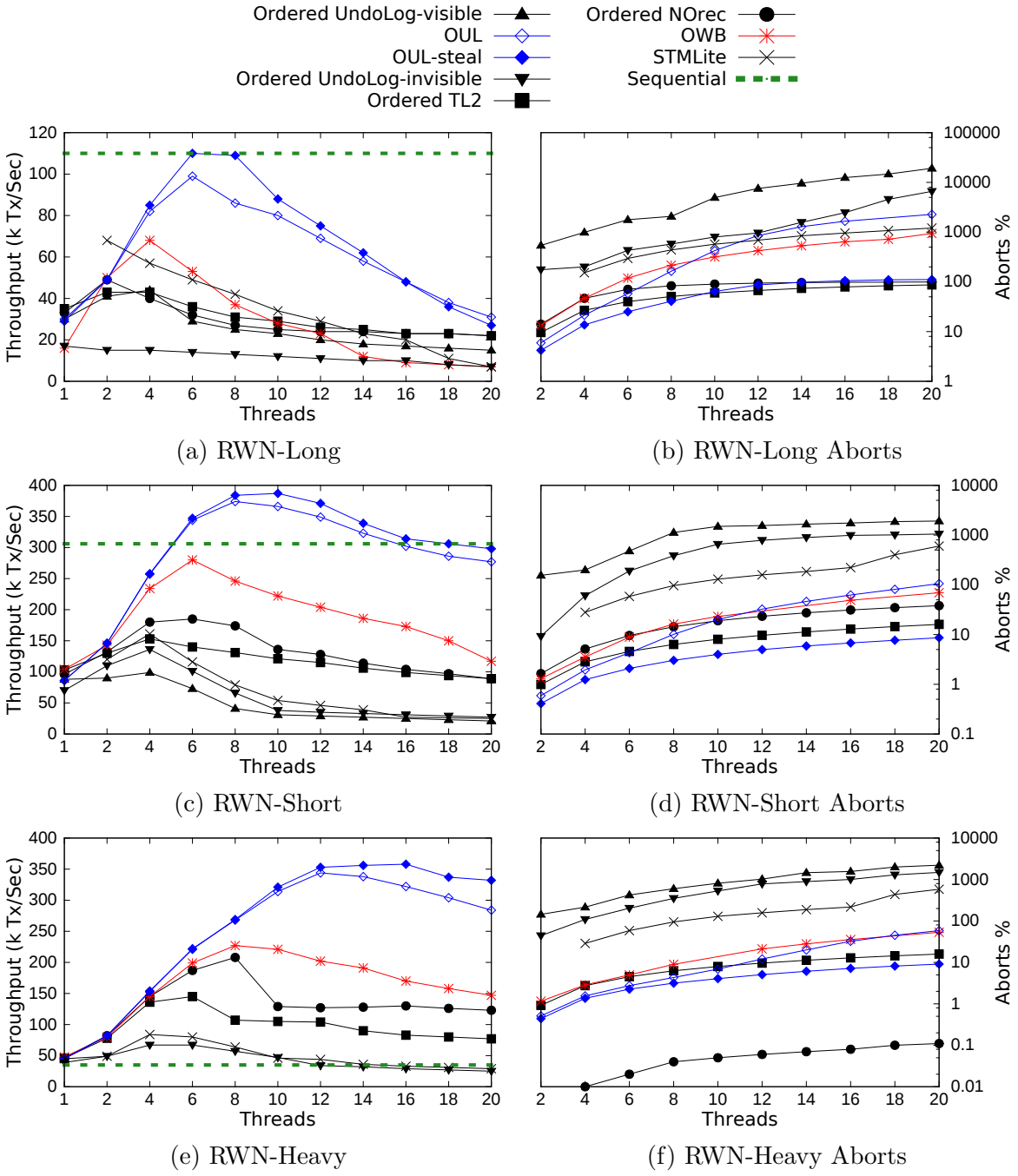
(c) RNW1-Short

(d) RNW1-Short Aborts

(e) RNW1-Heavy

(f) RNW1-Heavy Aborts

Figure 7.4: ReadNWrite1 Benchmark.



(a) RWN-Long

(b) RWN-Long Aborts

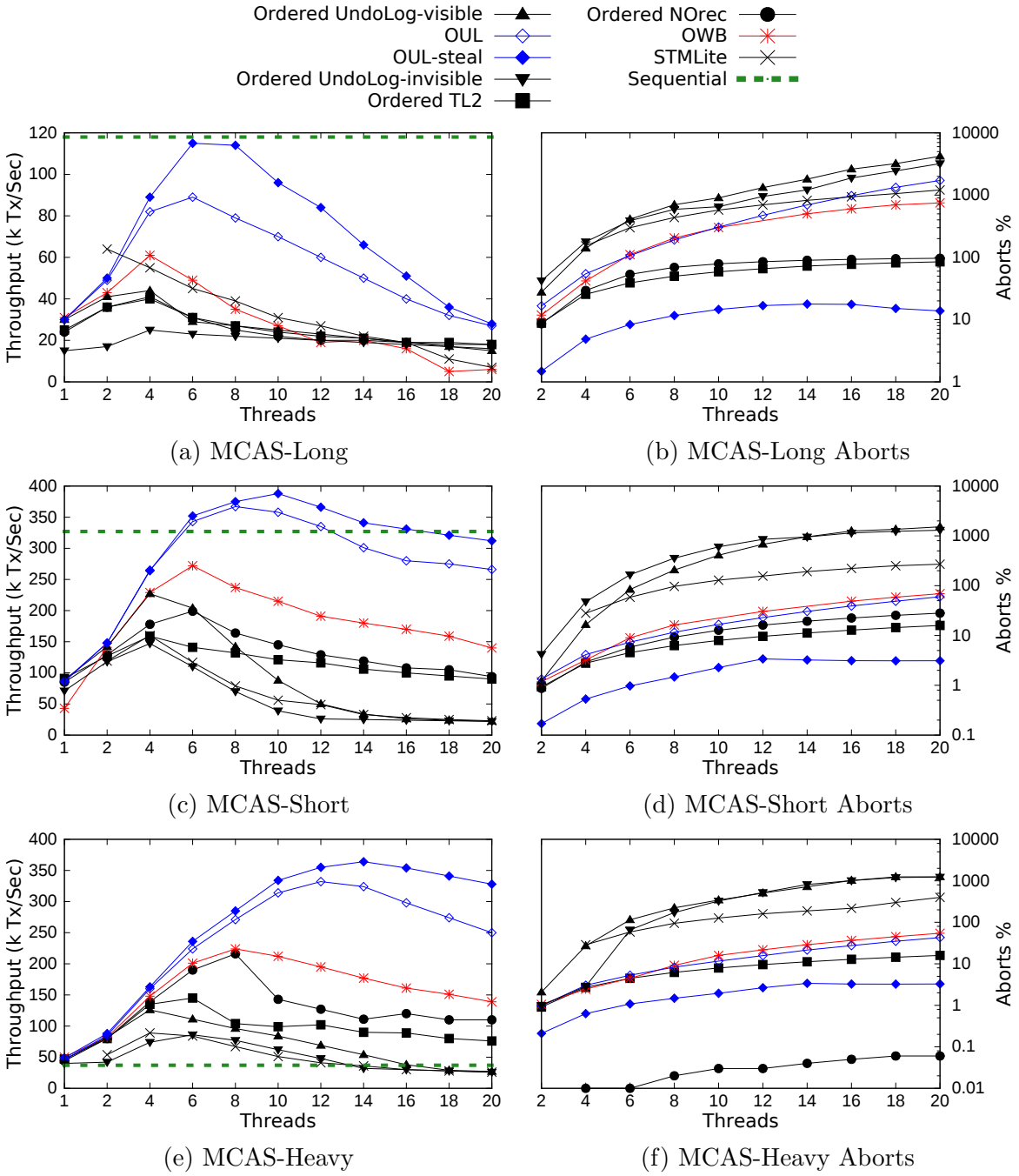
(c) RWN-Short

(d) RWN-Short Aborts

(e) RWN-Heavy

(f) RWN-Heavy Aborts

Figure 7.5: ReadWriteN Benchmark.



(a) MCAS-Long (b) MCAS-Long Aborts

(c) MCAS-Short (d) MCAS-Short Aborts

(e) MCAS-Heavy (f) MCAS-Heavy Aborts

Figure 7.6: MCAS Benchmark.

operations. In all configurations, OUL achieves the best throughput, while OUL-Steal suffers from the overhead of its lock management scheme without actually gaining from that, as the disjoint transactions do not have any shared accesses. UL-vis achieves a throughput near to OUL-Steal, thanks to the simplicity of its immediate write strategy. Given the absence of aborts, we can show the transactional access overhead for each of these algorithms. It is intuitive that UndoLog algorithms (including UL-vis, UL-inv, OUL, OUL-Steal) benefit from having the values already in memory, thus they outperform others. In fact, the UndoLog's main drawback is the costly abort, which never happens in this benchmark. With Long transactions (Figure 7.3a), STMLite benefits from eliminating lock usage at the write-back phase and it has minimal overhead at low numbers of threads. On the other hand, for Short transactions (Figures 7.3b and 7.3c) the Ordered TL2 algorithm performs better. OWB has a moderate overhead relative to the other write-back algorithms.

In *ReadNWrite1Bench* (Figures 7.4a-7.4f), the transaction write-set is very small, hence it implies a low number of aborts. Similarly, UndoLog algorithms excel here as well. With *long* and *heavy* transactions (Figure 7.4a, 7.4e), the processing done by workers is balanced with the validator overhead, so both OUL and OUL-Steal scales well with increasing the number of workers. On the other hand, the validator represents a performance bottleneck for short transactions (Figure 7.4c), resulting in a slightly lower scalability.

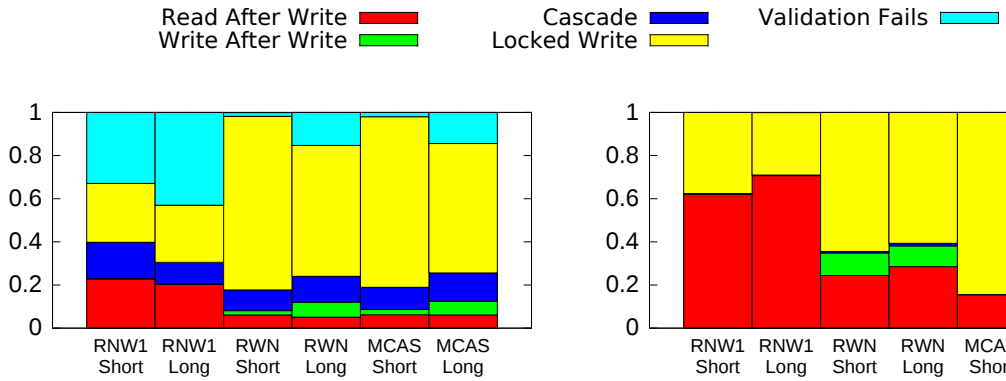
In *ReadWriteN* (Figures 7.5a-7.5f), the large transaction write-set introduces a challenge for both undo-log (increases the number of aborts) and write-buffer algorithms (delay at commit time). The cooperative execution enables OUL, OUL-Steal and OWB to outperforms all other algorithms at all workloads. OUL-Steal outperforms OUL by 10% because it significantly reduces the number of aborts (Figures 7.5b, 7.5d, and 7.5f.

Similar to ReadWriteN, *MCASBench* has a large write-set but the abort probability is lower than before because each pair of read/write acts on the same location. Figures 7.6a-7.6f illustrate the impact of increasing workers with the different workloads. We noticed a similar trend to *ReadWriteN*.

Analyzing The aborts reasons and breakdown

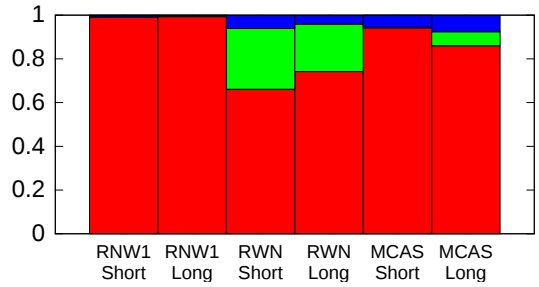
The breakdown of the abort reasons for OWB, OUL, and OUL-Steal is shown in Figure 7.7. Aborts are measured for the number of workers that achieved the maximum throughput.

In OWB (Figure 7.7a), with *RNW1bench* aborts due to validation failure represent the main reason; while in write-intensive benchmarks, such as *RWNbench* and *MCASbench*, aborts are mainly (65%-82%) due to concurrent commits (*Locked Write*). However, only 3% of these cases falls in WAW, which means that OWB can benefit from the lock-steal optimization and save a considerable amount of aborts. However, applying lock-steal on OWB would complicate the design and the validation procedure. The reason is that transactions use commit-time locking and rely on the version number to validate their read-set. With lock-

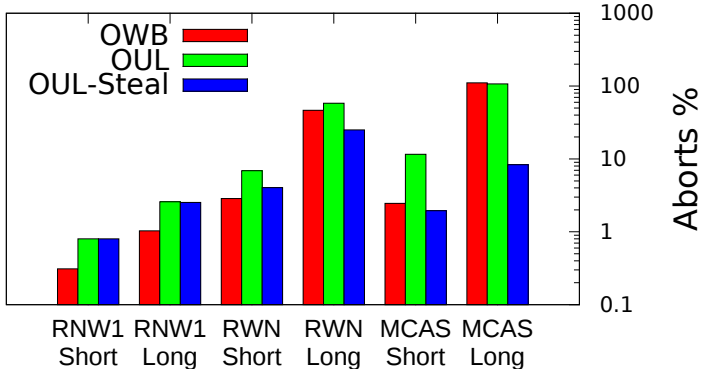


(a) OWB Algorithm

(b) OUL Algorithm



(c) OUL-Steal Algorithm



(d) Total Number of Aborts

Figure 7.7: Aborts Breakdown

steal, multiple writers would increment the version number, thereby readers would not be able to do the validation simply.

For OUL and write-intensive benchmarks, concurrent writes generate between 70% to 85% of total aborts; a WAW represents at most 10% of them. In OUL-Steal, stealing the lock eliminates the problem of concurrent writes, and narrows write-write conflicts to only the WAW anomaly. However, it introduces several changes to the abort characteristics: a writer transaction that steals the lock becomes able to abort any invalid speculative readers earlier than before. This was reflected on increasing the number of *Read After Write* aborts; the probability of triggering cascading aborts is increased if compared to OUL (Figures 7.7b, 7.7c); and the total number of aborts of OUL is reduced by one order of magnitude (Figures 7.4b, 7.4d, 7.5b, 7.5d, 7.6b, 7.6d, 7.7d).

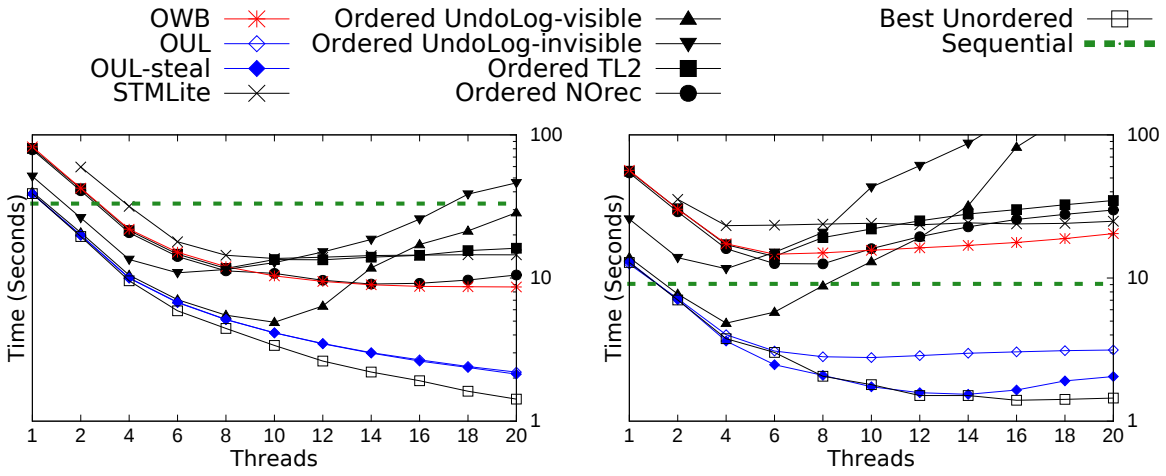
Although OUL-Steal substantially reduces the number of aborts, the speed-up is on average 20%. The reasons for that are: the abort procedure for OUL-Steal is longer than OUL (2-4× in our experiments) because it involves recursive rollback for stolen locks. This outweighs the reduction of the number of aborts; and OUL uses encounter time locking, thus aborts are detected at an early stage. This reduces the impact of aborting. In contrast, lazy algorithms (e.g., OWB) are greatly affected by aborts because the whole transaction needs to be re-executed given that the invalidation is detected at commit time. It is worth noting that, in OUL algorithms the abort cost differs according to the transaction type. In fact, aborting a write transaction requires restoring its original value, thus forcing the other transaction involved in the conflict to wait for the restoration of old written values; whereas aborting the readers is cheaper.

Figure 7.7d shows the total number of aborts in the maximum throughput scenario. OUL experiences higher aborts than OWB because of the eager accesses, while OUL-Steal avoids this drawback and experiences lesser, yet longer, aborts.

7.4.2 STAMP Benchmark

Stanford Transactional Applications for Multi-Processing (STAMP) [37] is a benchmark suite with applications covering a variety of domains (see Section 5.9.2). Figures 7.8 and 7.9 shows the execution time of the aforementioned algorithms (lower is better). Two applications (Yada and Bayes) have been excluded because they expose non-deterministic behaviors, thus their evolution is unpredictable. The datapoints for competitors that do not scale in some configuration are omitted to preserve the scale and readability of the plot. For completeness, we also included the performance of the unordered STM algorithm (among those in Figure 7.2) that behaves best in each plot.

In *Kmeans*, both OUL and OUL-Steal scale when increasing the number of workers, while under high contention OUL-Steal performs better (Figure 7.8b). OWB and Ordered NOrec have similar performance, but OWB does not degrade at high thread count.



(a) Kmeans Low (b) Kmeans High

Figure 7.8: Execution time of STAMP Kmeans (Y-axis log scale).

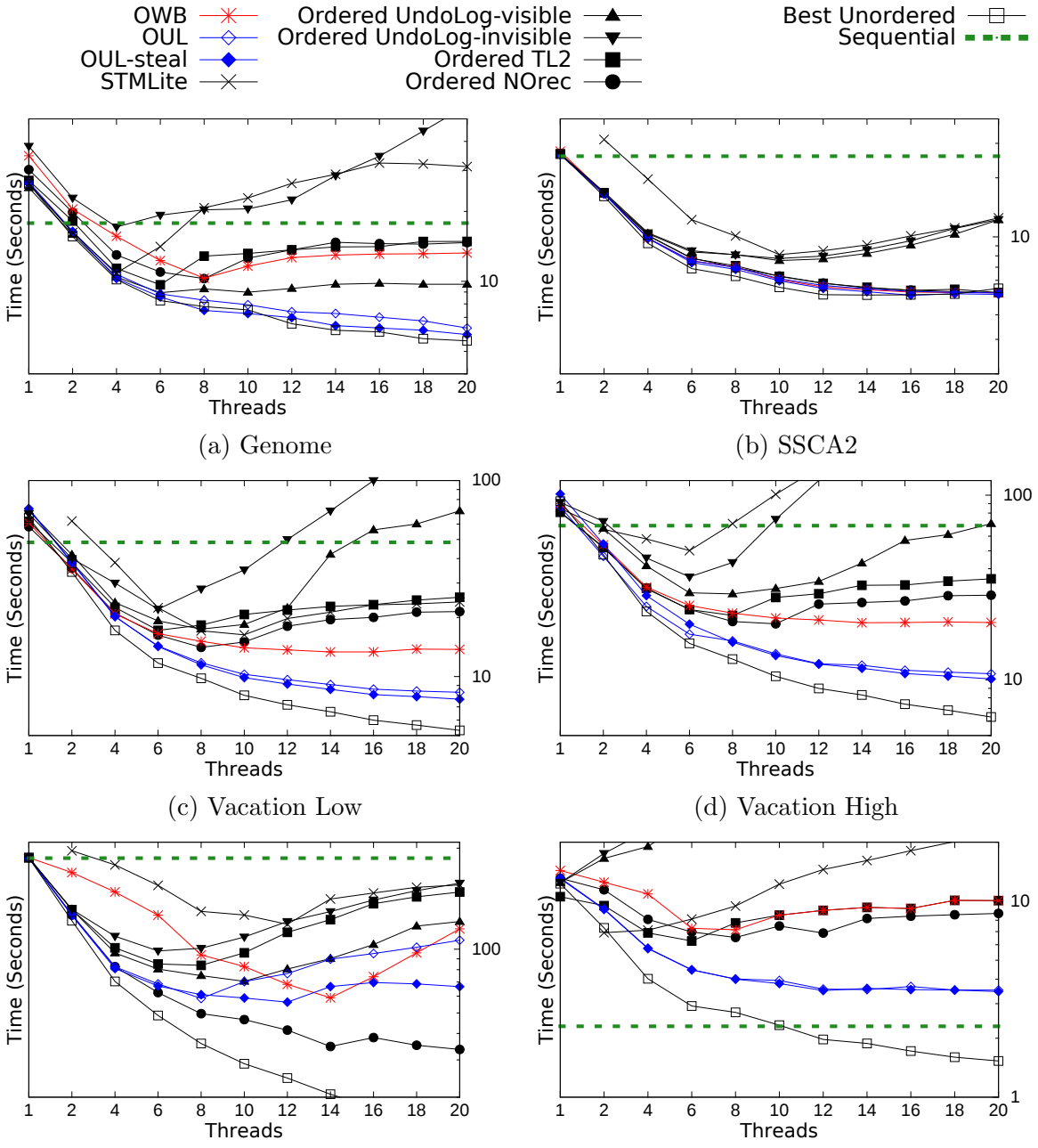
Genome exhibits a little contention which makes OUL and OUL-Steal perform similarly (Figure 7.9a).

The amount of contention in *SCAA2* is low as the large number of graph nodes leads to infrequent concurrent updates. Figure 7.9b shows that all algorithms perform almost equally and benefit from optimistic concurrency.

With *Vacation*, our cooperative model boosts the performance of the proposed algorithms, and they scale well when increasing the number of workers (clients) (Figures 7.9c and 7.9d).

Labyrinth is a multi-path maze solver. The maze is represented as a three-dimensional uniform grid, and each thread tries to connect input pairs by a path of adjacent maze points. Upon finding a path, it is is highlighted at a shared output grid. Transactions conflict when their paths overlap. In Figure 7.9e, NOrec outdoes other algorithms because of two reasons: *1)* as Labyrinth updates adjacent addresses for the path, it is prone to produce false sharing for all other algorithms that use locks; and *2)* NOrec employs a value-based validation, thus when two conflicting transactions updating a maze point with the same value, they commit successfully.

Intruder, a network intrusion detection system using signatures. It compares the captured packets against a dictionary of intrusion signatures. Packets are processed in parallel, grouped in sessions, and stored in a self-balanced (red-black) tree. Transactions guard the tree operations, and the contention is high and depends on the frequency of the rebalance operation. Figure 7.9f shows that not all algorithms scale well; besides, the sequential execution outperforms all of them (except the unordered).



(a) Genome

(b) SSCA2

(c) Vacation Low

(d) Vacation High

(e) Labyrinth

(f) Intruder

Figure 7.9: Execution time of STAMP applications (Y-axis log scale).

```
Begin
  Read last-committed transaction

Read X
  Search in write-set
  Or
  Read from memory if unlocked
  Or
  Read speculative value and
  add-to-dependency-list
  Or
  Abort higher age speculative write
  and read from memory
  if last-committed was changed
        Validate ReadSet

Write X
  Buffer write in write-set

TryCommit
  Validate read-set
  Acquire lock on write-set
  Expose values in write-buffer
  Validate read-set written locations

Commit
  Validate read-set
  Release locks

    --- METADATA ---
Read Set
Write Buffer
Dependency List
```

(a) OWB Algorithm

```
Begin
  Set Tx state as active

Read X
  Read from memory if unlocked
  Or
  Read speculative locked value
  Acquire read lock
  Or
  Abort higher age speculative write
  and read from memory

Write X
  Acquire write lock
  Backup value to the undo-log
  Write the value

TryCommit
  Set Tx state as pending

Commit
  Set Tx state as inactive

    --- METADATA ---
Undo Log
RW-Lock
```

(b) OUL Algorithm

Figure 7.10: OWB and OUL Algorithms Summary

7.5 Discussion

In the past two chapters, we presented OWB, OUL, and OUL-steal algorithms that effectively address the problem of committing transactions with an order defined prior to execution. Our results show that even if a system requires a specific commit order, it is possible to achieve high performance exploiting parallelism with data conflicts. Figure 7.10 summarizes the high level methods of the main two algorithms: OWB and OUL.

Chapter 8

Extending TM Primitives using Low Level Semantics

In order for a TM implementation to be generic, conflicts are usually detected at the level of memory addresses. For this reason, the TM abstraction can be expressed using four instructions: `TM_BEGIN`, `TM_END`, `TM_READ`, and `TM_WRITE`. The first two identify the transaction boundaries while the last two define the barriers for every memory read and write that occurs within those boundaries. TM algorithms differ in the way those instructions are implemented. Although frameworks may add other features, such as allowing external aborts, non-transactional reads/writes, or irrevocable operations, the above four instructions are used to form the body of most TM solutions.

Despite TM's high programmability and generality, its performance is still not yet as good as (or better than) optimized manual implementations of synchronization. To overcome that, researchers have investigated various approaches with different design choices. Regarding STM, they mostly varied the internal granularity of locking and/or validation of accessed memory addresses. Examples of those solutions include coarse-grained mutual exclusion of commit phases, as used in NOrec [53]; compact bloom filters [25] to track memory accesses, as used in RingSTM [174]; and fine-grained ownership records, as used in TL2 [59]. On the other hand, current HTM processors [149, 35] have a "best effort" nature because transactions are not guaranteed to progress in HTM (even if they are executed alone without any actual concurrency). That is why an efficient software "fallback" path is needed (i.e., hybrid TM) when hardware transactions repeatedly fail [61]. Recent literature proposes many compelling solutions that make the fallback path fast under different conditions [52, 152, 36, 61, 119].

The key commonality of all the aforementioned approaches is that they do not challenge the main objective of TM itself, which is providing generality at the application level. This is also the reason why those smart and advanced solutions still retain some of the fundamental inefficiency of TM. On the other hand, providing high performance in multi-threaded applications before the advent of TM, when thread synchronization was manually done using

fine-grained locks and/or lock-free designs, depended upon the specific application semantics. For example, identifying the critical sections and the best number of locks to use are design choices that can be made only after deeply knowing the semantics of the application itself (i.e., what the application does).

A related question that arises in this regard is: *Is there some room for including semantics in TM frameworks without sacrificing their generality?* If the answer is "yes", which is what we claim and assess in this chapter, then we will finally be able to overcome one of the main obstacles that has existed alongside TM since its early stages, and boost its performance accordingly. Recent literature provides a few semantic-based concurrency controls, which will be detailed in Section 8.1. However, they either solve specific application patterns [156], break the high abstraction of TM [92, 70], or are orthogonal to TM [90, 85].

Motivated by the above question, this chapter provides three major contributions. First, we identify a set of semantics that can be included in TM frameworks without impacting the generality of the TM abstraction (we call them *TM-friendly* semantics), and we extend the existing TM API to include such semantics. Second, we show how to modify STM algorithms to exploit such semantic-based APIs. Finally, we illustrate how we embedded those extensions in compiler passes (using GCC) so that the application developing experience will not be altered.

Regarding the first point, with TM-friendly semantics we mean those optimizations that can be decoupled from the application layer. In particular, this work focuses on optimizing conditional statements (e.g., `if x > 0`), and increments/decrements (e.g., `x++`), which are commonly used in legacy applications. More details about those semantics are presented in Section 8.2.

The second contribution involves deploying those TM-friendly semantics with existing state-of-the-art STM algorithms. Roughly, STM algorithms can be classified into two groups according to the technique used for validating transactions. The first group uses *version-based* validation, where each memory location keeps a version number that is used to identify memory changes. The second group uses *value-based* validation, where the content of each location itself is leveraged to detect memory modifications. For value-based algorithms, we propose *semantic validation* as a generalization of value-based validation, allowing TM frameworks to define a specific validator for the semantic-based instructions. For version-based approaches, we propose a methodology for adapting them to allow a hybrid (i.e., version/semantic) validation mechanism. In Section 8.3, we show how to modify NOrec [53] (a value-based algorithm) and TL2 [59] (a version-based algorithm) to include semantics. Then, in Section 8.4, we discuss the correctness of those new algorithms.

Our last contribution is to integrate semantic APIs and their corresponding STM algorithms into current TM frameworks. We propose two approaches to achieve that:

- The first approach is to implement semantic extensions entirely as a *compiler pass*, thus not exposing any API additions to the programmer. This approach has the

advantages of being entirely transparent and retaining backward compatibility with existing applications that leverage GCC's transactional API.

- The second approach involves exposing the new semantic APIs as TM interfaces. These new APIs give conscious programmers an opportunity to better exploit semantics while developing concurrent applications. Clearly, this approach increases the chance of achieving higher performance, with the cost of reducing, although marginally, the programmability level.

Since each of those two solutions fits specific interests, we assess both of them in this part. We assess the latter solution (which is easier to implement) by enriching the API of RSTM [116] framework with our semantics. Regarding the former, we show in Section 8.5 how we modified the compilation passes of GCC to provide full compilation support with limited overhead in terms of both compilation process and execution time.

In Section 8.6, we evaluated our semantic-based TM (using both RSTM and GCC) with the following applications: Bank, a benchmark that simulates a multithreading application where threads mostly perform money transfers; LRU-Cache, a benchmark that simulates a software cache with the least-recently-used replacement policy; a hash-table benchmark; and the STAMP benchmark suite [124]. The results show that enabling semantics boosts performance consistently, yielding a peak of 4× improvement when semantics is highly exploited. Also, contrasting the performance trend of GCC experiments with that of RSTM experiments allows understanding the consequences of moving the whole TM framework, including our semantic extensions, into the compiler level.

All the implementations used in this chapter, including the new version of GCC and RSTM, are available as open-source projects at: `https://bitbucket.org/mohamed-m-saad/extm`.

8.1 The Evolution of Semantic TM

Not surprisingly, the trials to include semantics in TM started in literature as early as TM itself. In fact, the potential objective of the first TM proposal, as it can be easily inferred from the title of the first TM paper [93], was providing architectural support for lock-free data structures. However, the approach proposed in that paper, as well as the subsequent approaches, was fairly general because its main objective was improving programmability. As a result, the performance of TM could not compete with handcrafted (i.e., very optimized) fine-grained and lock-free designs.

In the last decade, involving semantics to improve TM performance has been an important topic, addressed by approaches such as open nested transactions [136], elastic transactions [70], specialized STM [64], and early release [92]. The main downside of all those attempts is that they move the entire burden of providing optimizations to the programmer,

and propose a modified framework to accept those programmer modifications. Since TM has been mainly proposed to make concurrency control as transparent as possible from the programmers standpoint, the practical adoption of the above approaches remained limited. The innovations presented in this paper overcome those issues by providing solutions that preserve the generality of TM, do not give up optimizations and semantics, and cope with the current state-of-the-art TM implementations.

Another research direction focused on developing collections of transactional blocks (essentially data structure operations) that perform better than the corresponding "naive" TM-based counterparts (i.e., when the sequential specification of a data structure is made concurrent using TM). Methodologies like transactional boosting [90, 85], consistency oblivious programming [8, 14], semantic locking [72], and partitioned transactions [188] are examples of that direction. Despite the promising results, those approaches remain isolated from TM as a synchronization abstraction and appear as standalone components designed mainly for data structure.

Involving compilers in TM's concurrency control is currently becoming mandatory given the enhanced GCC release [180], which includes TM support. However, to the best of our knowledge, very few works addressed the issue of detecting TM-friendly semantics at compilation time similar to what we propose in this chapter. Among them, one recent approach proposes a new *read-modify-write* instruction to handle some programming patterns in TM [156]. However, that approach still addresses specific execution patterns and does not generalize the problem like our attempt, which rather pushes more in the direction of abstracting the problem and providing a comprehensive solution to inject semantics into existing TM frameworks.

8.2 TM-Friendly API

In this section we show the proposed semantics that can be injected into TM frameworks without hampering the generality of TM itself. As mentioned before, TM defines two language/library constructs for reading (`TM_READ`) and writing (`TM_WRITE`) memory addresses. In most cases, these constructs enforce a "conservative" conflict resolution policy; two concurrent transactions are said to be conflicting if they access the same address and at least one access is a write. Algorithm 9 gives an example that shows why such policy may be too conservative due to lack of semantics.

In this example, when T_1 executes its first line, existing TM algorithms save x and y in the read-set. Starting from this point, in order to preserve consistency, most TM implementations force T_1 to abort as soon as any concurrent change in x or y occurs. This abort can be triggered during the validation of T_1's next read (e.g., in NOrec), when T_1 tries to commit (e.g., in TL2), or immediately (e.g., in Intel HTM processors). In that specific example, since T_2 writes to x and y and commits before T_1 reaches its commit phase, most TM im-

Algorithm 9 Two transactions conflicting at the memory level but not at the semantic level.

Initially x = y = 5

TM_BEGIN(T_1)	
if x > 0 ‖ y > 0 **then**	
// Do reads/writes ...	
	TM_BEGIN(T_2)
	x++
	y- -
	TM_END
end if	
TM_END	

plementations force T_1 to abort. However, T_1 has no real issue at the semantic level and can safely commit since the boolean result of the conditional expression still holds, which means that the conflict triggered by the TM framework is a "false conflict" at the semantic level.

`TM_GT(address, value|address)`	greater than
`TM_GTE(address, value|address)`	greater or equals
`TM_LT(address, value|address)`	less than
`TM_LTE(address, value|address)`	less or equals
`TM_EQ(address, value|address)`	equals
`TM_NEQ(address, value|address)`	not equals
`TM_INC(address, value)`	increment
`TM_DEC(address, value)`	decrement

Table 8.1: Extended TM Constructs.

Examples like the above motivated us to design extensions to the traditional transactional constructs that enrich the TM programming model. Those constructs are classified according to their semantics into two categories (summarized in Table 8.1). The first category includes *conditional operators*, which take two operands and return a boolean state of the conditional expression. The operands in this category can be two addresses or an address and a value. At the memory level, a traditional execution of those constructs inside a transaction implies one or two calls to `TM_READ` (depending upon the type of operands). Using our constructs, we consider the whole expression as one semantic operation, and the safety of the enclosing transaction is preserved by validating that the return value of the condition remains the same until the transaction commits. The second category includes increment/decrement operations, which take an address and an offset as arguments. Unlike the first category, the traditional way of handling transactional increment/decrement involves both `TM_READ` and `TM_WRITE`. In our solution, leveraging semantics means invoking one semantic operation that performs the actual read only at commit time, which allows for more concurrency.

Including those semantic operations in TM frameworks is appealing for two reasons. First, they are commonly used in applications, as we show later with some examples. Second, the

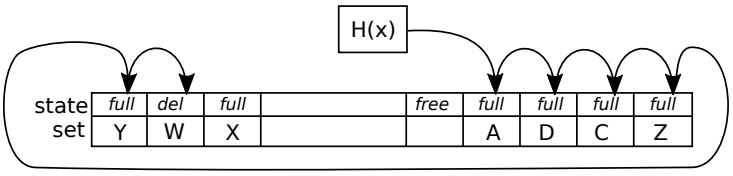


Figure 8.1: Probing a hash table with open addressing

integration can be entirely done at compilation time, where the compiler can detect semantic operators and translate them.

An interesting feature of the semantic operations listed in Table 8.1 is that they can compose by having more than one operator and/or more than one variable in the conditional expression. For example, the scenario shown in Algorithm 9 can be further enhanced if the whole conditional expression (i.e., `TM_READ(x) > 0 || TM_READ(y) > 0`) is considered as one semantic read operation. In this example, if the condition was initially `true` and then a concurrent transaction modifies only one variable, either x or y, to be negative, considering the clause as a whole avoids aborting T_1 given the `OR` operator. A similar enhancement consists of allowing complex expressions in conditional statements (e.g., `x + y > 0`), where modifications on multiple variables may compensate each other so that the return value of the overall expression remains unchanged. Although supporting such complex expressions is appealing because it may save additional aborts, integrating them into algorithm designs and GCC may add overheads in terms of compilation process and execution time. For that reason, we currently do not support those complex expressions, and we plan for further investigation on them. A more detailed discussion about those operations is in Section 10.2.2.

8.2.1 TM-friendly semantics in action

Algorithm 10 Using the semantic constructs to enhance hash table probing.

```
TM_BEGIN
        ▷ Using our constructs: while (TM_NEQ(states[index], FREE) && (TM_EQ(states[index], REMOVED) ||
TM_NEQ(set[index], value))
while TM_READ(states[index]) != FREE && (TM_READ(states[index]) == REMOVED || TM_READ(set[index]) != value)
do
    index = (index + probe)
end while
return TM_READ(states[index]) == FREE ? -1 : index;
TM_END
```

To further support the need of injecting semantics into the classical TM abstraction, we now show examples from real benchmarks and applications whose performance can be enhanced by our semantic TM extensions. These examples are clearly not exhaustive, but they are representative of programming patterns used in concurrent programming.

Hashtable with open addressing. Operations in such a hash table usually start by probing the table in order to find a matching index for a given hash value. Figure 8.1 depicts an example

of this probing. This function can be enhanced by our approach because it consists of a chain of conditional expressions that check specific semantics and do not impose certain values of `state` or `set` (e.g., it may only require the checked cells to be not free and either flagged as removed or having a different value from the hashed one). On the other hand, when using the classical read/write TM constructs, concurrent changes to the accessed cells (e.g., deleting 'D' and inserting 'B') will abort the probing transaction. Considering semantics through our proposed extensions avoid such aborts. Algorithm 10 depicts pseudocode of the probing method and its transformed semantic version.

Algorithm 11 Using the semantic constructs to enhance dequeue operation.

```
TM_BEGIN
                                          ▷ Using our constructs: If (TM_EQ(head, tail))
if TM_READ(head) != TM_READ(tail) then
    return false;
end if
item = array[TM_READ(head) % array_size];
                                          ▷ Using our constructs: TM_INC(head, 1);
TM_WRITE(head, TM_READ(head) + 1)
return true;
TM_END
```

Queues. Any efficient concurrent queue implementation should let an `enqueue` operation execute concurrently with a `dequeue` operation if the queue is not empty. However, this case is not allowed using traditional TM constructs because the `dequeue` operation compares the head with the tail in order to detect the special case of an empty queue. Algorithm 11 shows how we re-enable this level of concurrency in an array-based queue using our constructs.

Algorithm 12 Using the semantic constructs to enhance reservations in Vacation benchmark.

```
TM_BEGIN
for n = 0; n ¡ ids.length; n++ do
    res = tablePtr.find(ids[n]);
                                          ▷ Using our constructs: TM_GT(res.numFree, 0)
    if TM_READ(res.numFree) > 0 then
                                          ▷ Using our constructs: TM_GT(res.price, max_price)
        if TM_READ(res.price) > max_price then
            max_price = TM_READ(res.price);
            max_id = id;
        end if
    end if
end for
reservation = tablePtr.find(max_id);
                                          ▷ Using our constructs: TM_INC(res.numFree, -1)
TM_WRITE(res.numFree, TM_READ(res.numFree) - 1));
TM_END
```

Vacation. This application is included in the STAMP suite [124] and simulates a travel reservation system. The workload consists of clients' reservations; each client uses a coarse-grained transaction to execute its session. Vacation has two main operation profiles: making a reservation and updating offers (e.g., price changes). Although the reservation profile checks the common attributes of the offer (e.g., the number of free slots and the range

of price), most of those checks are semantic and do not seek specific values. Using the classical (more conservative) TM model, any update on offers will conflict with all concurrent reservations because of those conditional statements. Using our semantic extensions, as depicted in Algorithm 12, the reservation will not abort as long as the outcomes of the comparison conditions hold (e.g., `number of free slots > 0` and `price > max_price`). The key idea, which also explains well the intuition behind our proposal, is that a reservation does not use the exact value of price or the amount of available resources, it just checks if the price is in the right range and resources are still available.

Algorithm 13 Using the semantic constructs to enhance Kmeans benchmark.

```
TM_BEGIN
                                        ▷ Using our constructs: TM_INC(*new_centers_len[index], 1);
TM_WRITE(*new_centers_len[index], TM_READ(*new_centers_len[index]) + 1);
for j = 0; j < nfeatures; j++ do
                                        ▷ Using our constructs: TM_INC(new_centers[index][j], feature[i][j]));
    TM_WRITE(new_centers[index][j], TM_READ(new_centers[index][j]) + feature[i][j]));
end for
TM_END
```

Kmeans. Kmeans is another STAMP application, which implements a clustering algorithm that iterates over a set of points and groups them into clusters. The main transactional overhead is in updating the cluster centers, which can be enhanced using our *TM_INC* operation, as shown in Algorithm 13.

8.3 Semantic-Based TM Algorithms

The first step towards injecting semantics into STM algorithms is to find an abstract way to define them. The semantic operations listed in Table 8.1 can be seen as the implementation of two abstract methods:

```
bool cmp(operator, address, val)
    void inc(address, delta)
```

where `cmp` and `inc` represent the semantic actions that replace the normal TM behavior (`delta` can be positive or negative to support increment and decrement). In this abstraction, we restrict `cmp` operations in Table 8.1 to those that have an address and a value as arguments. However, as we show in Section 8.5, our compilation pass also detects the address-address case and translates it to a specific API call. Extending the STM algorithms presented in this section to cover the address-address case is straightforward, thus we do not include it to simplify the presentation.

In this section, we show how we integrate the above two abstract methods into two state-of-the-art STM algorithms: NOrec [53] and TL2 [59].

8.3.1 Semantic NOrec Algorithm (S-NOrec)

NOrec is an STM algorithm that exploits value-based validation to eliminate the need for fine-grained locks. A transaction stores the values it reads as a metadata in a local read-set and validates this read-set before every read, as well as at the commit time of writing transactions. The commit phase is protected by a single global timestamped lock. The validation procedure succeeds if all accessed addresses have the same values as what is saved in the read-set.

We extend NOrec to support our constructs as shown in Algorithm 14 (we call the new algorithm S-NOrec), mainly by executing `cmp` and `inc` using additional procedures. The main difference between `read` and `cmp` is that `read` appends the normal address/value pair to the read-set (line 47) while `cmp` saves the conditional expression (or its inverse if the condition is `false`) in the read-set (line 39). To simplify the `validate` procedure, we consider `read` as a semantic `TX_EQ` operation. Consequently, the `validate` procedure (lines 1-11) becomes a generalization of the original NOrec that uses a semantic validation instead of the original value-based one.

Both `read` and `cmp` read the address using a special `readValid` procedure (lines 12-19) that performs a read-set validation (if the global timestamp changed) to ensure the consistency of the current state of the read-set.

Supporting `inc` operations requires storing the delta (i.e., incremented or decremented value) in the write-set, and applying it at commit time. In practice, we support `inc` by overloading NOrec's write-set. In particular, a flag is added to each write-set entry to indicate whether it stores a standard write or an increment.

The cases where a variable is read/written (either semantically or non-semantically) by two different operations in the same transaction are handled by S-NOrec as follows:

- *write after write:* If an `inc` is preceded by a `write` or an `inc`, the new delta is accumulated over the entry's value without changing the entry's flag (line 52). If a `write` is preceded by a `write` or an `inc`, it just overwrites the value and changes the flag to indicate a `write` operation (line 58).
- *read after write:* Both `compare` and `read` check the write-set first for read-after-write conflicts (lines 35 and 44). If the write-set entry is an increment, the `inc` is promoted as a traditional `read` and `write` operation (see lines 22-24). The read part of the promotion is also done using the `readValid` procedure (line 22).
- *write after read:* This case is inherently covered because the value of the address will be validated anyway at commit time (because of the read) before the write takes place. It does not matter if the read/write operations are semantic or non-semantic.
- *read after read:* We add two different entries in the read-set for each read. Although this approach looks redundant and may nullify the gain of adopting a semantic validation

Algorithm 14 S-NOrec

```
1: procedure Validate(Transaction tx)
2:     time = global_lock
3:     if (time & 1) != 0 then go to 2 end if
4:     for each (addr, operation, val ) in reads do
5:         if ! (addr OP val) then                                  ▷ Semantic validation
6:             Abort()
7:         end if
8:     end for
9:     if time != global lock then go to 2 end if
10:    return time
11: end procedure

12: procedure ReadValid(Address addr, Transaction tx)
13:    val = *addr
14:    while snapshot != global lock do
15:        snapshot = Validate(tx)
16:        val = *addr
17:    end while
18:    return val
19: end procedure

20: procedure RAW(Address addr, Transaction tx)
21:    if writes[addr].type = INCREMENT then
22:        val = ReadValid(addr, tx)                                ▷ Promote increment
23:        reads.append(address, val, EQUALS)
24:        writes[addr] = ( entry.value + val, WRITE )
25:    end if
26:    return writes[addr].value
27: end procedure

28: procedure Start(Transaction tx)
29:    do
30:        snapshot = global_lock
31:    while (snapshot & 1) ≠ 0
32: end procedure

33: procedure Compare(Address addr, Operation op, Value operand, Transaction tx)
34:    if writes[addr] then
35:        return RAW(addr, tx) OP operand
36:    end if
37:    val = ReadValid(addr, tx)
38:    result = (val OP operand)
39:    reads.append(addr, operand, result ? OP : Inverse(OP))
40:    return result
41: end procedure

42: procedure Read(Address addr, Transaction tx)
43:    if writes[addr] then
44:        return RAW(addr, tx)
45:    end if
46:    val = ReadValid(addr, tx)
47:    reads.append(addr, val, EQUALS)
48:    return val
49: end procedure

50: procedure Increment(Address addr, Value delta, Transaction tx)
51:    if writes[addr] then
52:        writes[addr] = ( entry.value + delta, entry.type )
53:    else
54:        writes[addr] = ( delta, INCREMENT )
55:    end if
56: end procedure

57: procedure Write(Address addr, Value value, Transaction tx)
58:    writes[addr] = ( value, WRITE )
59: end procedure
```

if one read is semantic and the other is non-semantic, the overhead of discovering duplicates may not be negligible in the normal cases.

S-NOrec is the first STM algorithm, to the best of our knowledge, that supports `cmp` operations. For `inc` operations, a recent approach discusses supporting a pattern similar to our proposal [156]. Interestingly, in contrast with [156], S-NOrec maintains the same privatization and publication properties [121] of the original NOrec algorithm, since it still uses the global timestamp at commit time. In fact, there is no considerable overhead of S-NOrec over NOrec with respect to both processing time and memory occupied, as it only adds the read-set operation type and the write-set flag to the algorithm's metadata.

8.3.2 Semantic Transactional Locking 2 Algorithm (S-TL2)

TL2 is an STM algorithm that maps the shared memory locations to a table of ownership records (`orecs`). Writing transactions lock the `orecs` of their write-set entries at commit instead of acquiring a global lock as in NOrec. Because of that, writing transactions can commit concurrently as long as they access different `orecs`, and hence TL2 is known to scale better than NOrec. To validate reads, TL2 leverages: *i)* a global timestamp, which is atomically incremented by each writing transaction at commit; *ii)* a *start_version* for each transaction, which is set at the beginning of the transaction by snapshotting the global timestamp; and *iii)* an *orec_version* for each `orec`, which is modified by the writing transaction at commit time. This way, validation is done simply by ensuring that the *orec_version* of a newly read address is less than the *start_version* of the transaction, and revalidating the *orec_versions* of the whole read-set at commit time (only if the transaction is a writing transactions).

Algorithms 15 and 16 depicts our extended version of TL2 (called S-TL2). The write-set handlers (`inc`, `write` and `raw`) are similar to Algorithm 14, so we did not show them in Algorithm 15. On the other hand, supporting the `cmp` operation in S-TL2 is more complex than S-NOrec. The first issue is that the actual addresses and their values are not saved in the read-set; only the corresponding `orecs` are saved. To solve this problem, we first define a separate *compare-set* for saving `cmp` operations whose structure is similar to S-NOrec's read-set. In particular, a `read` operation saves the `orec` of the address in the read-set (line 59), and a `cmp` operation saves the actual address along with the information about the compare operation in the compare-set (lines 21 and 41).

The second problem is that we now have two ways for validating reads: the first relies on value-based validation (for `cmp` operations), and the second relies on the relation between the *read_version* of an `orec` and the *start_version* of the enclosing transaction (for `read` operations). To address this issue in an efficient way, we split the execution into three phases. The first phase starts from the transaction begin until the first `read` operation. The second one starts from the first `read` until right before commit. The last phase is the commit

Algorithm 15 S-TL2

```
1: procedure START(TRANSACTION tx)
2:     tx.start_version = global_timestamp
3: end procedure

4: procedure COMPARE( ADDRESS addr, OPERATION op,
5:                    VALUE operand, TRANSACTION tx)
6:     if writes[addr] then
7:         return RAW(addr, tx)
8:     end if
9:     orec = getOrec(addr)
10:    L1 = orec.version
11:    if tx.reads.isEmpty() then                          ▷ Phase 1: No reads yet
12:        if orec.lock ∉ {tx, φ} then
13:            go to 9                                     ▷ Wait until unlocked
14:        end if
15:        val = *addr
16:        L2 = orec.version
17:        if L1 ≠ L2 then
18:            go to 9                                     ▷ Retry read
19:        end if
20:        result = (val OP operand)                       ▷ Add to compare-set
21:        compares.append(addr, operand, result ? OP : Inv(OP))
22:        if L1 >start_version then
23:            time = global_timestamp
24:            ValidateCompareSet()
25:            if time != global_timestamp then
26:                go to 23                                ▷ Retry validation
27:            else
28:                start_version = time                    ▷ Extend start_version
29:            end if
30:        end if
31:    else                                                ▷ Phase 2: At least one pervious read occur
32:        if orec.lock ∉ {tx, φ} then
33:            Abort()
34:        end if
35:        val = *addr
36:        L2 = orec.version
37:        if L1 >start_version ∨ L1 != L2 then
38:            Abort()
39:        end if
40:        result = (val OP operand)                       ▷ Add to compare-set
41:        compares.append(addr, operand, result ? OP : Inv(OP))
42:    end if
43:    return result
44: end procedure

45: procedure READ(ADDRESS addr, TRANSACTION tx)
46:    if writes[addr] then
47:        return RAW(addr, tx)
48:    end if
49:    orec = getOrec(addr)
50:    L1 = orec.version
51:    if orec.lock ∉ {tx, φ} then
52:        Abort()
53:    end if
54:    val = *addr
55:    L2 = orec.version
56:    if L1 >start_version ∨ L1 != L2 then
57:        Abort()
58:    end if
59:    reads.append(orec)                                  ▷ Add to read-set
60:    return val
61: end procedure
```

Algorithm 16 S-TL2 - cont.

```
62: procedure ValidateReadSet(Transaction tx)
63:     for each (orec) in tx.reads do
64:         if orec.lock ∉ {tx,φ} ∨ orec.version ¿ start_version then
65:             Abort()
66:         end if
67:     end for
68: end procedure

69: procedure ValidateCompareSet(Transaction tx)
70:     for each (addr, operation, val ) in tx.compares do
71:         current = *addr
72:         orec = getOrec(addr)
73:         if orec.version ¿ start_version then
74:             if orec.lock ∉ {tx,φ} then
75:                 repeat until orec.lock = φ                    ▷ Wait until unlocked
76:             end if
77:             if !(current OP val) then                         ▷ Semantic validation
78:                 Abort()
79:             end if
80:         end if
81:     end for
82: end procedure
83: procedure Commit(Transaction tx)
84:     AcquireWriteSetLocks(tx)
85:     time = global_timestamp
86:     if start_version ≠ time then
87:         ValidateCompareSet(tx)
88:     end if
89:     if !CAS(global_timestamp, time, time+1) then
90:         go to 85                                              ▷ Retry compare-set validation
91:     end if
92:     if start_version + 1 ≠ time then
93:         ValidateReadSet(tx)
94:     end if
95:     WriteBack(tx, time + 1)
96:     ReleaseWriteSetLocks(tx)
97: end procedure
```

phase.

In the first phase, before the first `read` operation, `cmp` operations can be optimized similar to S-NOrec (lines 11-30): the transaction's *start_version* is not used, rather the compare-set is validated after each `cmp` operation (line 24). If this validation succeeds, the transaction's *start_version* is extended (line 28). This way, we allow semantic validations as long as no `read` operation is executed yet. Another optimization, although less important, is when the address's `orec` is observed to be locked by a concurrent transaction. In this case, the `cmp` operation waits until the `orec` is unlocked instead of aborting the transaction (line 75). In those cases, we employ a timeout mechanism (not shown in the algorithm) to avoid starvation. This optimization makes sense only for `cmp` operations. This is because for `read` operations, observing a locked `orec` means that its *orec_version* will likely be updated before it is unlocked, which also means that the read will be invalidated and the transaction will be aborted anyway. The same optimization is made when a concurrent transaction changes the *orec_version* while reading the variable (line 18), and also when the global-timestamp is

changed during the compare-set validation (line 26).

In the second phase, after the first `read` operation, `cmp` operations have to preserve consistency with previous reads, and therefore the transaction's *start_version* cannot be extended anymore. That is why, in this phase, both `read` (lines 45-61) and `cmp` (lines 31-42) validate that the newly read address (even if inside a `cmp` operation) is consistent with the previous reads, by comparing the *orec_version* with the transaction's *start_time* as the original TL2 does.

The commit phase is depicted in lines 83-97. In TL2, the commit phase starts by locking the writes' `orecs` and atomically incrementing the global timestamp. Then the reads are re-validated. If validation succeeds, writes are published and then locks are released. The commit phase of S-TL2 differs from that of TL2 in two points: *i)* the way reads are validated; *ii)* the way the global timestamp is incremented.

Regarding the first point, the read-set and the compare-set are validated differently using `ValidateReadSet` (lines 62-68) for the former and `ValidateCompareSet` (lines 69-82) for the latter. Specifically, when the *read_version* of an `orec` is greater than the *start_version* of the transaction, which means that the value of the address may have been changed, `ValidateReadSet` aborts the transaction (line 65), while `ValidateCompareSet` re-computes the expression and aborts only if the return value changes (line 78).

Second, if a concurrent transaction starts its commit phase during `ValidateCompareSet`, the compare-set has to be re-validated. This is important because the return value of one `cmp` operation may be affected by this new commit, and thus `ValidateCompareSet` procedure may return an incorrect result. Lines 85-90 depict how we achieve that (the order of lines is important here): the global timestamp is snapshotted, `ValidateCompareSet` is called, and then the global timestamp is incremented using `CAS` instead of `AtomicFetchAndAdd`. If the `CAS` fails, validation is retried. It is worth to note that this mechanism is not needed for `ValidateReadSet` because it conservatively aborts the transaction if any `orec` in the read-set has changed.

S-TL2 requires adding a compare-set as well as a flag in the write-set in addition to the orignal metadata of TL2. An additional source of overhead is that `cmp` operations may involve calling `validateCompareSet`, whose execution time is linear with respect the size of the compare-set itself. However, as we show in the evaluation, those overheads are mostly dominated by the performance gain due to avoiding unnecessary aborts.

8.4 Correctness

The correctness of a TM algorithm is usually inferred by proving that all histories it generates are *opaque* [76]. We infer the correctness of S-NOrec and S-TL2 in the same way.

We start by roughly recalling some definitions related to opacity, borrowed from [76], to

make the presented intuitions self-contained. A history $\mathcal{H}$ is a sequence of operations issued by transactions on a set of shared objects. Intuitively, we say that a history $\mathcal{H}$ is *sequential* if no two transactions are concurrent. A sequential specification of a shared object *ob*, called *Seq(ob)*, is the set of all sequences of operations on *ob* that are considered correct when executed sequentially. A sequential history is *legal* if, for every shared object *ob*, the subsequence of operations on *ob* in $\mathcal{H}$ is in *Seq(ob)*. Two histories are *equivalent* if they contain the same transactions with the same operations and the same return values. Given a history $\mathcal{H}$, $Complete(\mathcal{H})$ indicates the set of histories obtained by committing or aborting every commit-pending transaction in $\mathcal{H}$, and aborting every other live transaction in $\mathcal{H}$. A history $\mathcal{H}$ is *opaque* if any history in $Complete(\mathcal{H})$ is equivalent to a legal sequential history $\mathcal{S}$ that preserves the *real-time order* of $\mathcal{H}$.

The definition of opacity is general enough to be applied on shared objects with generic APIs, as long as every shared object has a well-defined *sequential specification* based on those APIs. However, as we mentioned before, most TMs consider the *read-write register* abstraction with two APIs: a `read` operation, which takes no argument and returns the current state of the register, and a *write* operation, which takes a value v as argument and returns always *ok*. This simple scheme implies a trivial sequential specification for each register x, as defined in [76]: *the set of all sequences of read and write operations on x, such that every read operation returns the value given as an argument to the latest preceding write operation (the initial value is the default).*

Although this simple scheme best fits shared memory models, it does not match the APIs of our *semantic-based* TM because it does not distinguish between reads/writes that are made within a comparison/increment expression and all other reads/writes. That is why the first step towards proving the correctness of our algorithms is to define the new abstraction for our TM. Our TM algorithms (S-Norec and S-TL2) implement a TM whose shared *register* export four operations: `read`, which takes no argument and returns the current state of the register; `write`, which takes a value v as an argument and returns always *ok*; `inc`, which takes a value d as an argument and returns always *ok*; and `cmp`, which takes a value *v_cmp* and an operator type *Op* whose value is given from the enum {==, !=, ¿, ¿=, ¡, ¡=} as arguments and returns *true* or $false$.

The sequential specification of a register x in our TM, $Seq(x)$, is defined as follows: the set of all sequences of `read`, `write`, `cmp`, and `inc` operations on x, such that in each them:

- every `read` operation returns $v + \sum d$, where v is the value given as an argument to the latest preceding `write` operation, w, and $\sum d$ is the sum of the values given as arguments to every `inc` between the `read` operation and w;
- every `cmp` operation returns the boolean value of the expression (v *Op* *v_cmp*), where v is the return value of the corresponding `read` operation.

Algorithm 17 gives an example that clarifies the importance of defining a new abstraction

Algorithm 17 A history that is opaque with our APIs.

```
                                        Initial values are 0
TM_BEGIN(T1)
if x >= 0 then
                                                    TM_BEGIN(T2)
                                                    x = 1
                                                    y = 1
                                                    TM_END
   z = y;
end if
TM_END
```

for our TM. Using the original *read-write register* abstraction, the corresponding history has two possible equivalent sequential histories ($T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$), and both of them are not legal because T_1 returns an illegal value for y in the former and an illegal value for x in the latter. However, using our (correct) abstraction, $T_2 \rightarrow T_1$ is an equivalent legal sequential history because x is read using `cmp` and the return value of this `cmp` is legal, which means that the history is opaque.

Another interesting example to assess the correctness of S-NOrec and S-TL2 is shown in Algorithm 18. The history of this example is not opaque even with the new APIs because the value of x at the moment of executing the `cmp` operation was different from its value when the transaction read y.

Algorithm 18 A history that is not opaque with our APIs.

```
                                        Initial values are 0
TM_BEGIN(T1)
z = y
                                                    TM_BEGIN(T2)
                                                    x = 1
                                                    y = 1
                                                    TM_END
if x >= 1 then
   z = 1
end if
TM_END
```

Based on the two cases in Algorithms 17 and 18, it is easy to understand the idea behind proving opacity of histories containing `cmp` operations, which is proving that: *i)* the address read inside each `cmp` is consistent with all the previous reads at the moment of computing the return value of the conditional expression, and *ii)* this return value does not change until the transaction commits (even if the value of the address becomes inconsistent).

Proving that a history containing `inc` operations is opaque is easier to infer. This is because the read part of `inc` can be deferred to the commit phase where the address is locked, and thus the whole `inc` operation is considered as a `write` operation during the transaction execution. The only exception for that is when the address accessed by an `inc` operation is also accessed by another operation in the same transaction. In the following two sections, we show how those cases are covered by S-NOrec and S-TL2.

8.4.1 Correctness of S-NOrec

Based on the opacity definition, the correctness of S-NOrec can be inferred if we identify the *legal* sequential history $\mathcal{S}$ that is equivalent to a generic history $\mathcal{H}$ generated at any point of its execution (after completing $\mathcal{H}$). Roughly, $\mathcal{H}$ may contain committed transactions (either read-only or writing) and live transactions (considering aborted transaction as live right before they trigger the abort call). $\mathcal{S}$ is identified, similar to NOrec, as follows: committed writing transactions are serialized when they CAS the global timestamp at commit; and both read-only and live transactions are serialized when the validation of their last finished `read`/`cmp` succeeds (i.e., after the committed writing transaction that sets the global timestamp with the value returned at line 10).

The history $\mathcal{S}$ remains legal with the existence of `cmp` operations because of the following reasons: *i)* every address is initially read consistently using `readValid` procedure in all `read`, `cmp`, and `RAW` calls; and *ii)* the semantic validation made after each read and during commit guarantees that the return values of each `cmp` remain the same.

The existence of `inc` operations also does not affect the legality of $\mathcal{S}$ because: *i)* at commit time, `inc` is handled exactly like `write`, which is safe because the transaction has an exclusive access to the address (in fact commit phases in NOrec are executed serially); and *ii)* live transactions that execute `inc` along with other operations on the same address are always consistent because the read operations (`read` and `cmp`) check the write-set first and promote the `inc` operation if needed, and the write operations (`write` and `inc`) properly override the write-set entry.

8.4.2 Correctness of S-TL2

The legal equivalent serialization of a history generated by S-TL2 is slightly different from TL2. In fact handling `inc` operations does not affect the serialization because of the same reasons mentioned for S-NOrec. However, we identify two differences that arise due to `cmp` operation.

First, committed writing transactions are serialized in TL2 when they atomically increment the timestamp. In S-TL2, this atomic increment is replaced with a `CAS` operation which forms the new serialization point (line 89). Using `CAS` instead of `AtomicFetchAndAdd` is needed because it is not legal for a transaction to observe the writes of any transaction that increments the global timestamp after it. Note that it is guaranteed that the transaction observes the writes of all transactions that increment the timestamp before it, because the `orecs` of the write-set entries are locked before incrementing the timestamp (line 84), and the transaction wait until those `orecs` are unlocked.

Second, the serialization of read-only and live transactions depends on the phase they are executing. If the transaction is in the first phase, before any `read` operation, the serialization

_ITM_S2R*type*	address-address semantic read operation
_ITM_S1R*type*	address-value semantic read operation
_ITM_SW*type*	semantic write operation

Table 8.2: Extended GCC ABI.

point is similar to S-NOrec, when the validation during the last `cmp` operation succeeds (more specifically, when *start_version* is advanced at line 28). That is because all the committed writing transactions so far did not change the return value of all `cmp` operations. On the other hand, if the transaction is in the second phase, after the first `read` operation, it is serialized similar to TL2, namely before all writing transactions that commit with a timestamp greater than its *start_version*. That is legal because: *i)* all `cmp` operations in the first phase are consistent up to the current value of *start_version*; and *ii)* all `read` and `cmp` operations in the second phase use this *start_version* in validation.

8.5 Integration with GCC

GNU Compiler Collection (GCC) supports STM since version 4.7 and HTM since version 4.9. The integration resulted in adding `_transaction_atomic` to the constructs of C/C++ [7, 112]. GCC translates statements within a `_transaction_atomic` block to the appropriate TM calls that follow an Application Binary Interface (ABI) similar to the TM ABI proposed by Intel [5]. Those calls are handled according to the TM algorithm chosen by the programmer. The implementation of those TM algorithms is encapsulated in the *libitm* library[1].

The first, straightforward, step we made towards embedding our semantic interfaces is adding three semantic operations to *libitm*'s ABI (see Table 8.2). The first two operations, `_ITM_S2R` and `_ITM_S1R`, handle `cmp` operation with address-address and address-value modes, while `_ITM_SW` handles `inc` operation. Then, we deployed our S-NOrec algorithm as an additional TM algorithm in the *libitm* library, and implemented the new ABI operations as described in Section 8.3.1. Due to lack of a TL2 implementation that matches the baseline we used to construct our S-TL2, we plan the integration of S-TL2 as a future work. Besides, in the TM algorithms currently existing in *libitm* library, those new operations are implemented by delegating their execution to the classical read and write handlers.

The next, more complicated, step is to detect the code patterns of our semantic operations (`cmp` and `inc`) during compilation. We did that after GCC generates the GIMPLE [122] representation of the program. GIMPLE is a language independent, tree-based representation that uses 3-operands expressions (except for function calls) in the Static Single Assignment

[1]We use GCC 5.3 and *libitm* libraries from `https://github.com/mfs409/transmem`, which include NOrec.

(SSA) [50] form. We chose GIMPLE representation to deploy our optimization passes for two reasons. First, GIMPLE is both architecture and language independent, thus optimizing it is considered a transparent *middle-end* optimization. Second, GIMPLE uses temporary variables to put its expressions in a 3-operands form, where every variable is assigned only once. This form simplifies the dependency analysis.

The `tm_mark` pass is one of the optimization passes on the GIMPLE representation where statements that perform transactional memory operations are replaced with the appropriate TM built-ins. We extended this pass to detect the code patterns of `cmp` and `inc` operations as follows.

- *`cmp`:* For any conditional expression we track the origins of its two operands along the GIMPLE tree. If one origin refers to a direct transactional memory access and the other refers to either a literal value or a local variable, then we replace the condition with a call to the `_ITM_S1R` built-in. If the two origins refer to direct transactional memory accesses, we use `_ITM_S2R`.

- *`inc`:* For any transactional write, we track the origin of its right hand side, and if it is calculated using a mathematical "+" or "-" equation, we track both its operands. If the origin of one of them is a transactional read to the same written address, and the origin of the second operand is either a literal value or a local variable, we call the `_ITM_SW` built-in instead of generating the transactional write. At this stage, the original transactional read of the `inc` still exists and has to be removed. We did so by developing another optimization pass, named `tm_optimize`, that removes TM read calls for *never-live variables*, since the read part of every `inc` becomes one of those *never-live variables* reads after replacing the write part with our call. This pass is made in a conservative way; it does not remove a read if there is no guarantee that it is *never-live*.

A side optimization that our `tm_optimize` pass performs is removing any TM read that is part of a never-live assignment, even if it is not originally part of an `inc` operation. The current GCC version does not perform any *liveness optimization* or *dead assignment identification* on the transactional code, therefore it does not remove such reads.

Another important note is that, the case when a shared variable is involved in both semantic (i.e., `cmp` and `inc`) and non-semantic (i.e., `read` and `write`) operations of the same transaction are handled by the TM algorithm as mentioned in Section 8.3 (see lines 35 & 52 of Algorithm 14, and line 7 of Algorithm 15), therefore it is not needed to detect those cases in the compiler passes.

One of the advantages of our optimizations is that they reduce the number of TM calls of both `_ITM_S2R` and `_ITM_SW` from two to one. Such reduction has a visible impact on application performance; in fact, TM calls are costly because GCC performs three indirect calls per TM call. Also, our pass does not require a complex alias analysis for tracking the

	Hashtable		Bank		LRU	
	base	*semantic*	*base*	*semantic*	*base*	*semantic*
Read	3440	0	22.5	0.05	173	12
Write	6.2	6.2	12.7	0	19.7	19.7
Compare	-	3440	-	10	-	161
Increment	-	0	-	12.7	-	0.03
Promote	-	0	-	0.05	-	0.01

	Vacation		Kmeans		Labyrinth		Yada		SSCA2		Genome		Intruder	
	base	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*	*base*	*semantic*
Read	14704	13714	25	0	176	4	142	135	2	1	84	84	28.5	28.5
Write	28.5	12	25	0	173	173	21.4	21.4	2	1	3	3	2.6	2.6
Compare	-	989.5	-	0	-	172	-	7	-	0	-	0.06	-	0
Increment	-	16.7	-	25	-	0	-	0	-	1	-	0.01	-	0
Promote	-	15.7	-	0	-	0	-	0	-	0	-	0	-	0

Table 8.3: Average Number of Operations per Transaction.

operands origin, because we look for simple expression patterns that are usually reside in the same basic block.

8.6 Evaluation

We tested our extended semantic-based TM on a set of micro-benchmarks, as well as applications of the STAMP suite [124]. We conducted our experiments on an AMD machine equipped with 2 Opteron 6168 CPUs, each with 12-core running at 1.9 GHz. The total memory available is 12 GB. We reported the throughput for the micro-benchmarks, and the application execution time for STAMP, by varying the number of threads executing concurrently. We reported the results for both RSTM and GCC implementations.

Table 8.3 shows the average number of invocations per operation type in the used benchmarks. They are measured at runtime using RSTM because it provides more flexibility to extract statistics than GCC. In this table, semantic and non-semantic algorithms are contrasted to give an intuition about the number of read and write operations saved by applying our semantic constructs. Reduction is substantial, which enables performance improvement as showed later.

8.6.1 RSTM-based implementations

In the following experiments, throughput and abort rate were computed for NOrec and TL2 in both their semantic and original (i.e., non-semantic) versions.

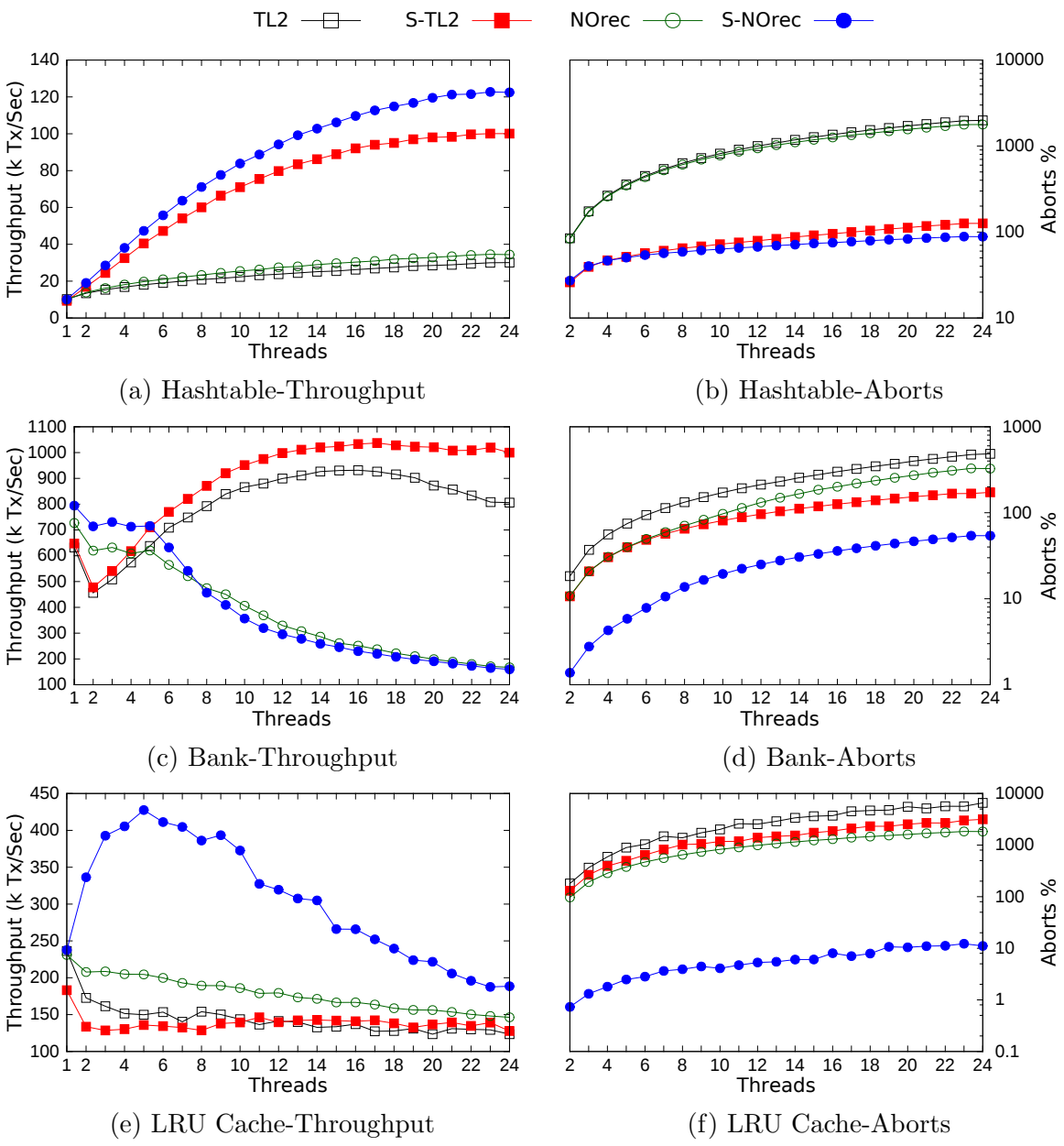
Micro Benchmarks

In our first set of experiments we considered three micro benchmarks: Hashtable with Open Addressing, Bank, and Least Recently Used (LRU) Cache.

Hashtable with Open Addressing. The workload in this experiment was a collection of *set* and *get* operations, where each transaction performed 10 set/get operations. Both S-NOrec and S-TL2 exploited our semantic extensions in the probing procedure, as depicted in Algorithm 10. As a result, and as shown in Table 8.3, all `read` operations were transformed into semantic `cmp` operations. This reduced the number of aborts by one order of magnitude (Figure 8.2b), which directly raised the throughput (3.5× speedup) in both algorithms (Figure 8.2a).

Bank. Each transaction performs multiple transfers (at most 10) between accounts with an overdraft check (i.e., skip the transfer if account balance is insufficient). In the semantic version of the benchmark, the reads/writes were transformed into `cmp` and `inc` operations. As shown in Figure 8.2c, exploiting semantics helps S-NOrec to outperform NOrec at low-contention (1-8 threads). However, when contention increases, both NOrec and S-NOrec degrade and perform similarly. This is mainly because the probability of having true conflicts increases, and transactions start to abort even if they are semantically validated. In TL2, concurrent commits are allowed, thus it scales better than NOrec. Similarly, S-TL2 benefits from the underlying semantics and performs 20% better than TL2, and incurs 2.5× fewer aborts.

LRU Cache. This benchmark simulates an $m \times n$ cache with least-frequently-used replacement policy. The cache uses m cache lines, and each line contains n buckets. Each bucket stores both the data and the hit frequency. Each transaction either sets or looks up multiple entries in the cache. Table 8.3 shows that 93% of the `read` operations were transformed into `cmp` operations. Accordingly, as shown in Figures 8.2e and 8.2f, S-NOrec reduced the aborts by two order of magnitude and achieved up to 2× speedup. S-TL2 was not improved much (only 25% speedup). The reason is that the non-transformed reads in S-TL2 prevented it from advancing its snapshot (i.e., it makes the first phase described in Section 8.3.2 shorter); thus, any compare operations had to preserve the snapshot identified by the *start_version* and the overall behavior becomes similar to TL2.



(a) Hashtable-Throughput

(b) Hashtable-Aborts

(c) Bank-Throughput

(d) Bank-Aborts

(e) LRU Cache-Throughput

(f) LRU Cache-Aborts

Figure 8.2: Micro Benchmarks using RSTM.

STAMP

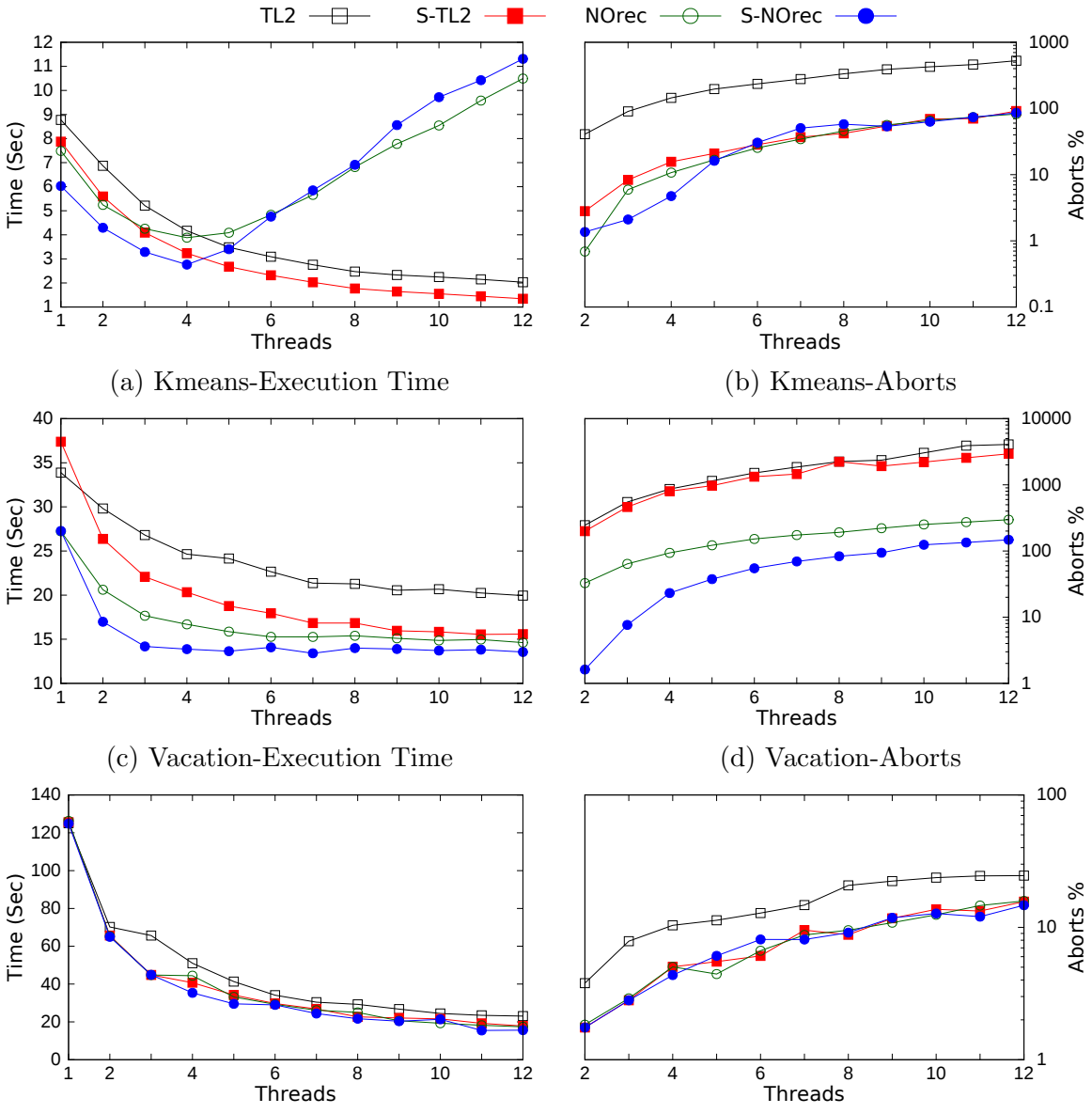
STAMP is a suite of applications designed for evaluating in-memory concurrency controls, for more details see Section 5.9.2. Figure 5.19 shows both execution time and abort rate in some of the STAMP benchmarks. We did not show the results of three applications (`Genome`, `Intruder`, and `SSCA2`) because we found that the semantic operations per transaction were very limited (see Table 8.3) and hence there was no difference in either abort rate or throughput. We also excluded `Bayes` because of its nondeterministic behavior. Since the performance saturated at a high number of threads in all tested applications, we show the results only up to 12 threads.

`Kmeans`. As illustrated in Algorithm 13, updating the centroid is changed by transforming all `writes` into `increments`. S-NOrec and S-TL2 achieve 25%-40% speedup (Figure 8.3a). However, at a high number of threads both NOrec and S-NOrec saturate and start to degrade in performance, which indicates a high contention workload due to the coarse-grained locking. Consequently, starting from 8 threads, S-NOrec slightly performs worse than NOrec (see Figure 8.3a), because it adds an overhead that is not exploited to reduce the abort rate (see Figure 8.3b).

`Vacation`. The reservation procedure was optimized as in Algorithm 12; however, only 7% of the reads were transformed into compares. This is because most of the `read` operations are part of the internal red-black tree operations. Additionally, almost all the `inc` operations were promoted to `read` and `write` operations because of an additional sanity check performed by the transaction. Although these two factors limited the gain of using the benchmark semantics, both S-NOrec and S-TL2 consistently outperformed the original algorithms.

`Labyrinth`. Different checks along the routing path (e.g., isEmpty, isGarbage) were transformed into semantic `cmp` operations, which allowed S-TL2 to outperform TL2 by 20%-50% speedup, and to reduce the aborts by 2× (see Figures 8.3e & 8.3f). Both S-NOrect and NOrec perform similarly, which indicates that transactions that fail in NORec's value-based validation also fail in S-NOrec's semantic validation. In [155], an optimized version of Labyrinth was proposed, where some non-transactional operations (memory copy) are moved outside the transaction, which in effect reduces the transaction size. Figures 8.4a & 8.4b show the performance in this new version. Although S-TL2 still has lower abort rate, the returned gain in performance became insignificant because most of the work became outside transactions.

`Yada` is a mesh triangulation benchmark implementing Ruppert's algorithm. Threads iterate over the mesh and try to produce a smoother one by identifying triangles whose minimum angle is below some threshold. NOrec's behavior is similar to `Labyrinth`. Interestingly, although S-TL2 reduced the number of aborts by 3×, throughput was not affected (Figures 8.4c & 8.4d). Our measurements revealed that the reason for this behavior is in the aborted transactions. Although resolving the semantic conflicts of transactions in S-TL2 allowed them to proceed with execution, true conflicts caused most of them to abort later. Therefore, the length of the aborted transactions in S-TL2 became longer than TL2 without



(a) Kmeans-Execution Time

(b) Kmeans-Aborts

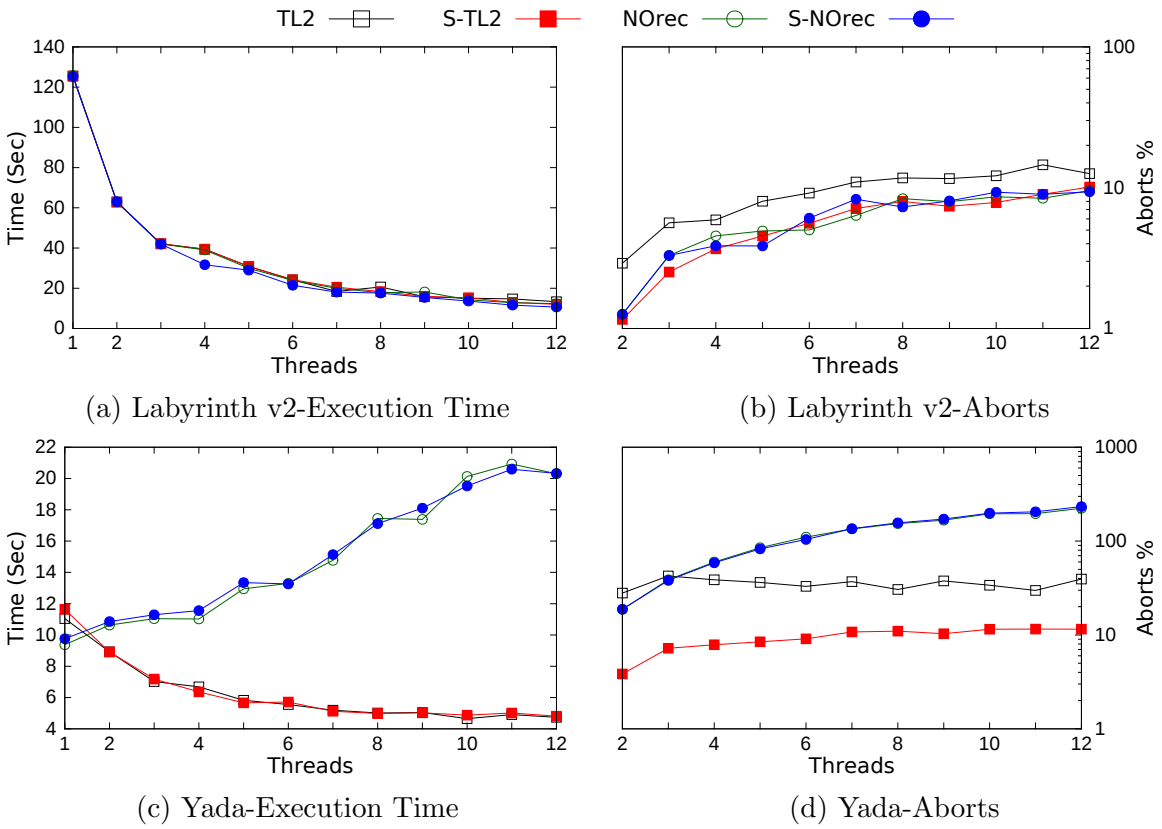
(c) Vacation-Execution Time

(d) Vacation-Aborts

(e) Labyrinth v1-Execution Time

(f) Labyrinth v1-Aborts

Figure 8.3: STAMP Applications using RSTM.



(a) Labyrinth v2-Execution Time

(b) Labyrinth v2-Aborts

(c) Yada-Execution Time

(d) Yada-Aborts

Figure 8.4: STAMP Applications using RSTM.

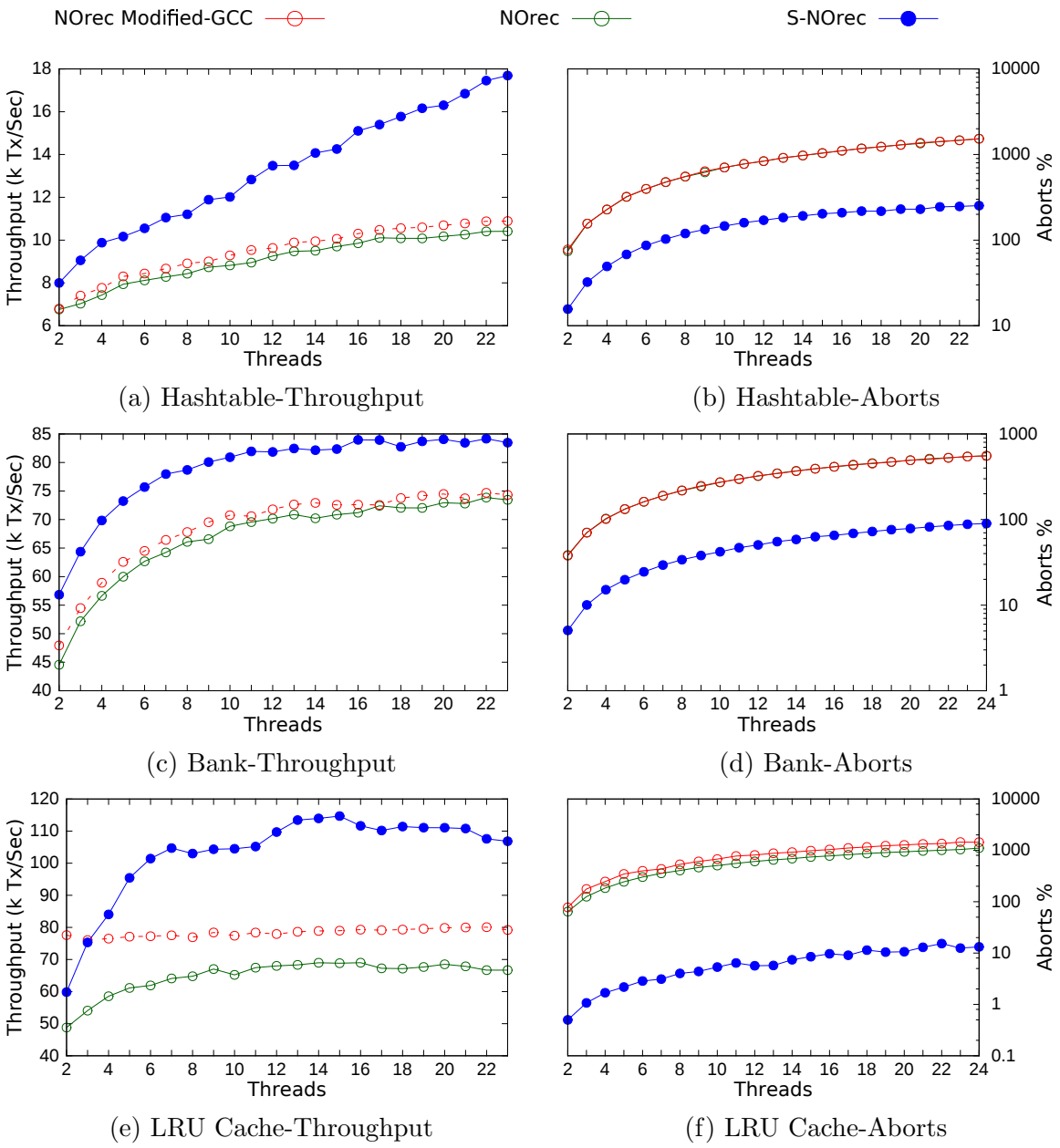
real benefit (since transactions eventually aborted). This is similar to what happened in Bank.

8.6.2 GCC-based implementations

Now we discuss the results of the above experiments using our modified GCC instead of RSTM. As mentioned in Section 8.5, in these experiments we focus on NOrec and S-NOrec. We also added one more version of NOrec that uses our modified GCC APIs but does not handle them semantically. The only difference between this version and NOrec is that, in the former, the applications calls our semantic APIs and internally delegates them to the normal reads/writes of NOrec, while the latter calls NOrec's APIs directly.

The results (Figures 8.5 and 8.6) follow the same trends described above with RSTM (we excluded `Labyrinth` and `Yada` from the GCC figures because NOrec and S-NOrec behave similarly in both of them, as shown in Figures 8.3 and 8.4). As before, our semantic extensions help improving performance in all benchmarks. Compared to RSTM, the actual throughput values decreased. This is mainly because GCC speculates every read and write

within the `_transaction_atomic` blocks, while RSTM speculates only addresses accessed using its transactional `TM_READ` and `TM_WRITE` APIs. However, GCC algorithms scale better than RSTM algorithms, mainly because of the internal optimizations in GCC, such as using more efficient structures to store and handle metadata. Interestingly, using our modified GCC, even without exploiting semantics ("NOrec Modified-GCC"), we observe some performance improvement due to decreasing the overall number of TM calls.



(a) Hashtable-Throughput

(b) Hashtable-Aborts

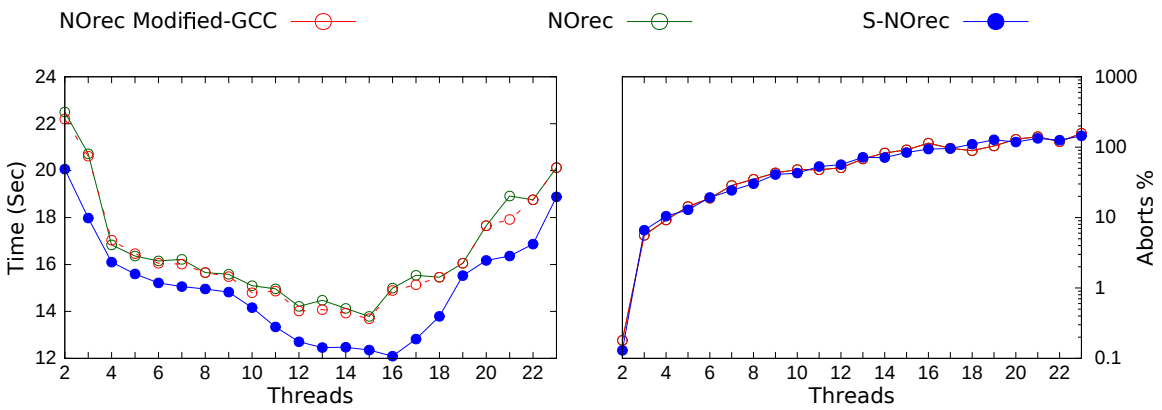
(c) Bank-Throughput

(d) Bank-Aborts

(e) LRU Cache-Throughput

(f) LRU Cache-Aborts

Figure 8.5: Micro Benchmarks using GCC.



(a) Kmeans-Execution Time

(b) Kmeans-Aborts

(c) Vacation-Execution Time

(d) Vacation-Aborts

Figure 8.6: Some STAMP Applications using GCC.

Chapter 9

Exploiting Hardware Transactional Memory

Intel has recently introduced Haswell [149] as the first mainstream CPU with transactional memory support. Haswell supports best-effort hardware transactional memory (HTM) using Restricted Transactional Memory (RTM) [97]. RTM eliminates the transactional loads and stores overhead, but it introduces hardware restrictions on the transaction size and the overall progressiveness. In this chapter, we propose novel techniques for executing transactions according to a predefined commitment order using Intel's RTM. First, we present two hardware algorithms: *Burning Tickets* and *Timeline Flags*. Next. we propose a hybrid hardware extension for our OWB algorithm (See Chapter 6), named OWB-RH. .

9.1 Haswell's RTM

Haswell introduces TM interface with three new instructions: XBEGIN, XEND, and XABORT. XBEGIN and XEND define the transaction start and end, while XABORT enables the transaction to abort itself. Transaction changes are stored in the processor cache, and at the commit time the cache is flushed to the main memory which provides the illusion of atomicity. Consequently, 1) any cache request (read or write) from another processor aborts the current transaction, 2) conflicts granularity is per cache-line, and 3) transaction is limited to the cache size and it can abort due to the capacity constraint. The conflict resolution policy implemented in Haswell is *requestor wins*, which means that there is no guarantee for forward progress. A transaction is aborted when its underlying processor receives either a coherence message (read or write) for a cache line in its write-set, or a coherence write message for one of its cache lines of the read-set. To overcome the absence of progress, a fallback software path is used after the transaction is repeatedly aborted. However, with the chance of having a transaction running in the fallback software path, it is required to

also coordinate the conflicts between transaction executing in the software path and in the hardware path [52, 36, 119].

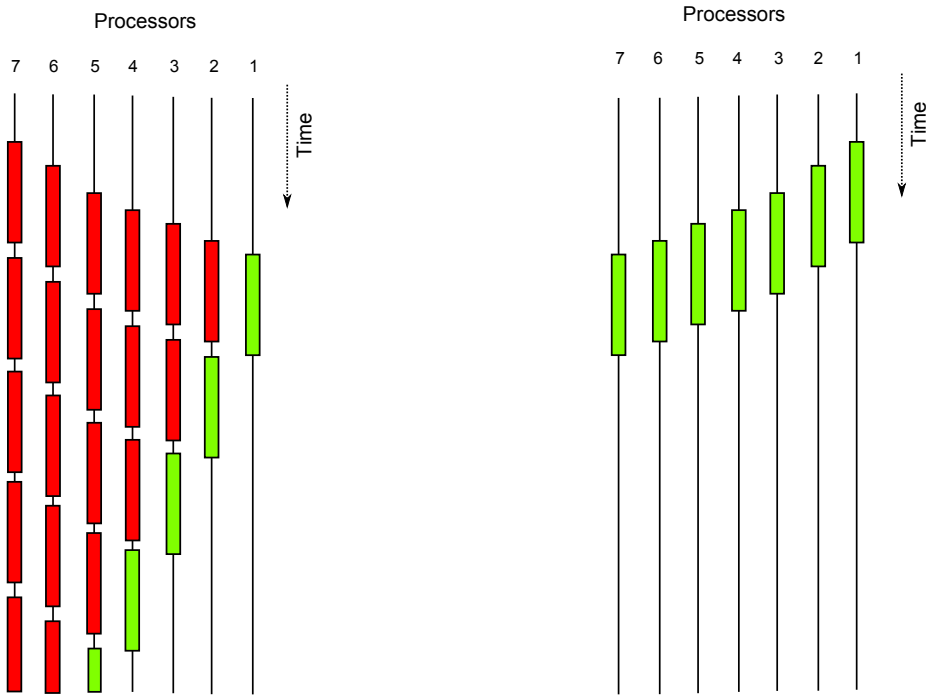
9.1.1 The Challenges of ACO Support

Ordering transactions is challenging using RTX, because it imposes information transfer between transactions for organizing the commitment order. Under RTX, any information transfer between transactions within their context implies aborting one of them. To illustrate this, let us assume there is a shared variable that stores the `next-to-commit` transaction's age. This variable will be checked when a transaction attempts to commit, and updated if the ACO matches its commit order. Let T_i be a transaction that is about to commit. If $i =$ `next-to-commit` then it will commit successfully. However, when $i \neq$ `next-to-commit` then the address of the `next-to-commit` shared variable is now loaded in the processor cache, hence, it becomes a part of the transaction read-set. Eventually, the system will update the `next-to-commit`, whenever the transaction with an age equals to `next-to-commit` commits. According to the *requestor wins* policy, transaction T_i (and any transaction that has read `next-to-commit`) has to be aborted. Thus, an algorithm that uses a shared variable such as the `next-to-commit` to identify the commitment order must abort (and retry) the transaction immediately when its age is not equal to `next-to-commit`. Aborting a transaction may severely degrade the performance because after multiple aborts, the transaction may switch to the slow fallback software path; meanwhile, with some implementations (e.g., SGL [189]) that could stop executing all hardware transactions. Unfortunately, RTM does not support escape actions (i.e., executing non-transactional access within the transaction), and will not support that in the near future. Using the escape action, transactions would be able to organize the commitment without aborting each other.

Figure 9.1 illustrates the worst and the best case scenarios for a `next-to-commit` based hardware algorithm. Recall that reading the `next-to-commit` occurs only at the end of the transaction.

9.2 The Burning Tickets Hardware Algorithm (BTH)

We propose a new technique, *burned tickets*, that reduces the number of aborts due to the ACO. The Burning Tickets algorithm (BTH) uses pure hardware transactional memory, and uses the time as a dimension for transferring the ordering information. In our technique, each transaction is associated with a list of *tickets*. A ticket holds a bi-value: true or false, and it the tickets are distributed over different cache lines. When the transaction completes its execution, it iterates over (burning) its ticket in order with a predefined delay between the accesses, and it checks if one of them has a true value. Upon finding a ticket with a true value, the transaction stops iterating and commits. If all the tickets were inspected and none



(a) Worst Case Scenario (b) Best Case Scenario

Figure 9.1: Executions of Next-to-Commit Hardware Algorithm

of them was true, the transaction aborts and restarts (using the same allocated tickets).

The transaction with the absolute minimum age does not have tickets (or have tickets that are all set), so it commits as soon as it finishes the execution. After its commit, the transaction iterates *in a reverse order* over the tickets, with the same delay, of its *immediate successor* transaction (i.e., with an age that is one higher than the current one). This strategy is twofold. First, it allows a committed transaction to notify its successor to commit its values without aborting it (as long as this happen before it consumes all tickets). Tickets that were set by the committed transaction do not affect the active transaction because they are not added *yet* to its read set. On the other hand, the committed transaction will not be affected by the coherence read message because setting the ticket is done out of the transaction. Second, if a higher age transaction conflicts with a lower age transaction, then it will enter the ticket burning phase, which will delay it from restart and aborting it again.

9.2.1 Configuring Tickets Burning

In this section we analyze the proposed procedure to determine the best configuration (e.g., number of tickets, the checking delay) that reduces the aborts probability, and the committing delay. Assuming the number of tickets per transaction is N; the delay between tickets

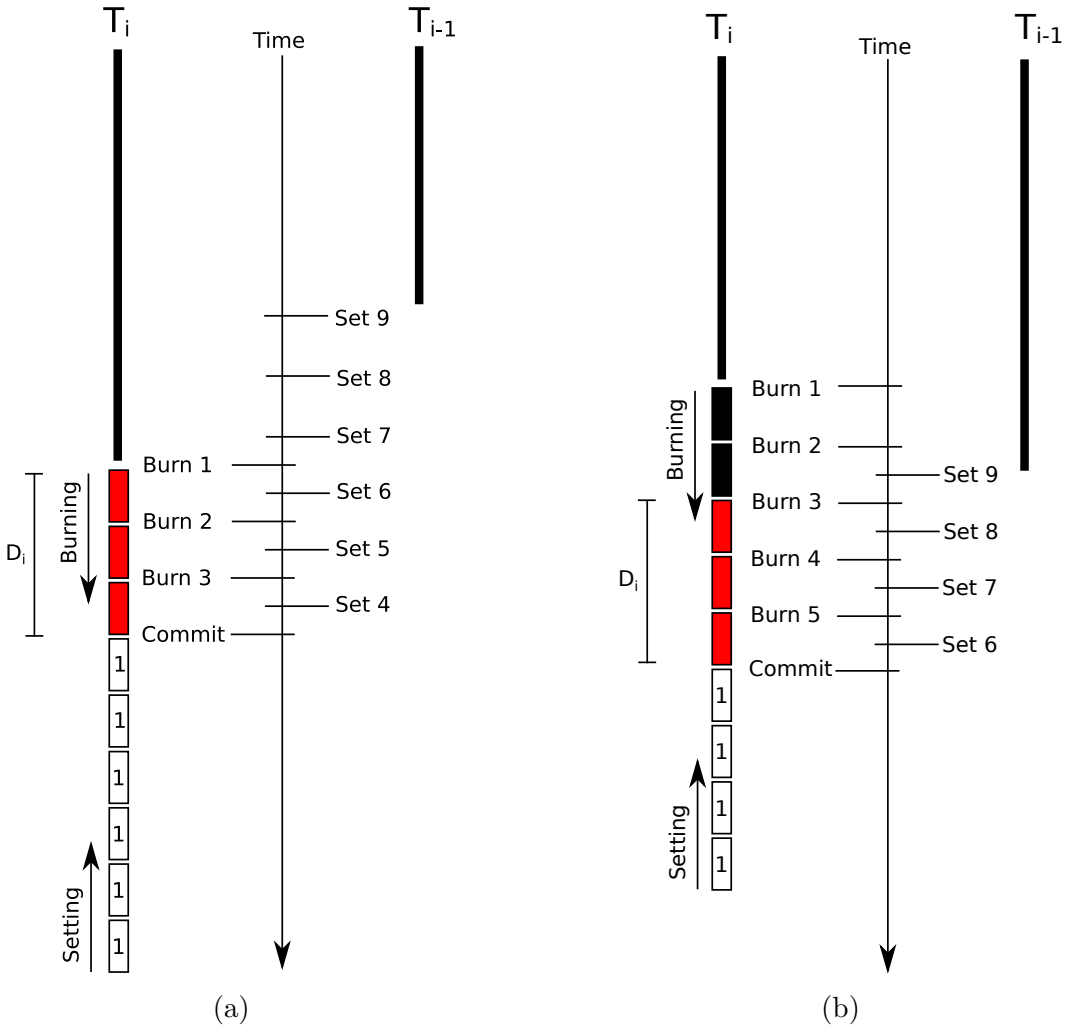


Figure 9.2: Tickets Burning with number of tickets (N)=9, and delay (D)=3

accesses (burning or setting) is d; the absolute clock time of committing the transaction with age i is t_i; the delay introduced by the algorithm to guarantee the in order commitment is D_i. Figure 9.2 illustrates an execution with four transactions (T_1-T_4).

The delay introduced for a transaction after it finishes the execution is the sum of: 1) the waiting time until its correct commit time, and 2) the time required for the transaction to be notified from its predecessor, which we denotes as D. In case T_i finishes before its predecessor T_{i-1} (see Figure 9.2a), it will burn $\lceil (t_{i-1} - t_i)/d \rceil$ tickets before T_i starts its notification phase. The notification requires reaching a mid-way ticket between the two transactions (T_i and $T_i - 1$), as they are moving in reversed directions. Therefore, we can represent D_i using the following relation.

$$D_i = (N - \lceil (t_{i-1} - t_i)/d \rceil)/2$$

Now, let us assume T_i finished after T_{i-1} (see Figure 9.2b). In this case, T_{i-1} will set the values of $(t_i - t_{i-1})/d$ before T_i starts burning the tickets. Similarly, we can calculate the mid-way ticket using the same way.

However, the transaction commit time of the predecessor needs to consider also the delay introduced by the the predecessor of the predecessor, which was waiting to commit. Therefore we need to substitute the value of t_{i-1} with $t_{i-1}+D_{i-1}$. So we can rewrite the general relation between the delay and the number of tickets using the following formula.

$$D_i = (N - \lceil |t_{i-1} + D_{i-1} - t_i|/d \rceil)/2, \text{ where } i > 1, \text{ and } D_1 = 0$$

Note that, a negative delays indicates that either the transaction runs out of tickets and will be aborted (when $t_{i-1} > t_i$), or the predecessor transaction set all tickets and the next transaction will commit as soon as it finishes ($t_{i-1} < t_i$).

That way a transaction can estimate the number of tickets based on the earlier times and delays, hence we can develop an *adaptive* model in which transactions controls the aforementioned parameters to reduce aborts.

Finally, a possible optimization is that the transactions can maintain a `next-to-commit` variable out of the transaction boundaries. A transaction updates it after it commits and does a check on its value *before* it starts. If the transaction `age` matches the `next-to-commit`, then it can skip the tickets checking and commits directly.

9.2.2 Conflict Resolution

As mentioned earlier, the default conflict resolution in Haswell's RTM is the requestor wins policy. In effect, this does not preserve the age-based priority imposed by our model. Recall that transactions cannot share online information while they are executing. In order to resolve this situation, we assume that a transaction can define the set of written addresses before execution. Thereby, the transaction constructs a *signature* [25] of its modifications ahead of the execution. Within execution and before any access (read or write), a transaction must inspect the signatures of all higher priority transactions (i.e., with lower age than the current) that are actively running, and it aborts itself (using XABORT) if a match found.

To reduce the signature bits that are set, we can construct in terms of the cache lines instead the addresses. Two addresses that are mapped to the same cache line could be considered as a single element, because they have the same cache invalidation effect. This optimization can easily be implemented in the signature hash function itself.

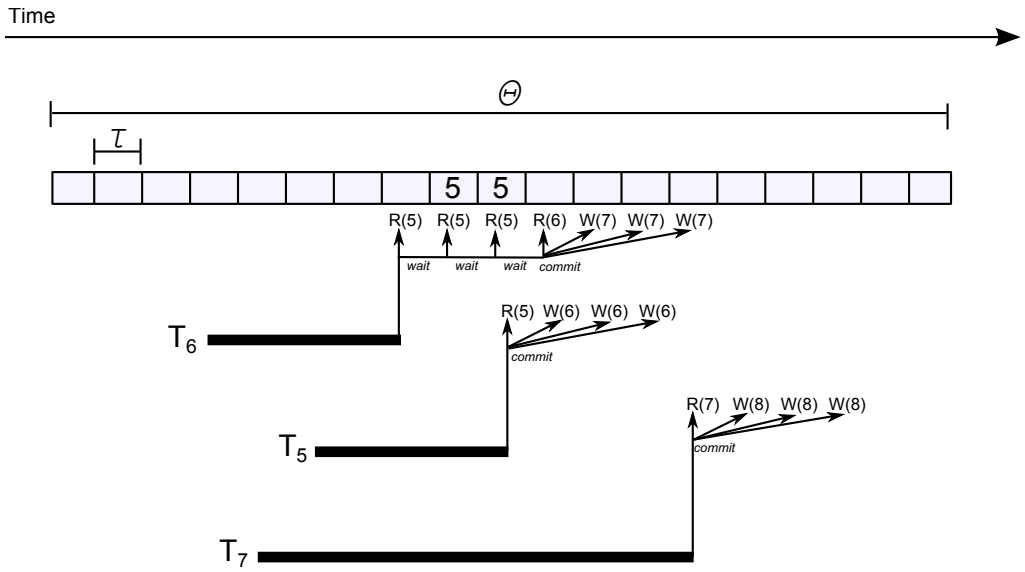


Figure 9.3: Timeline Flags with N=19, and M=3

9.3 The Timeline Flags Hardware Algorithm (TFH)

A drawback of the BTH algorithm is the usage of a per-thread meta-data (i.e., tickets). This limits the scalability of the algorithm, with larger meta-data the abort probability due to cache capacity increases. In this section, we propose a new algorithm, *Timeline Flags*, that uses a constant size of meta-data, that is independent of the number of active transactions (i.e., processor cores). Similar to BTH algorithm, we exploit the time as a dimension for exchanging the required ordering information.

Assume Θ is the length of the context switch time. Hardware transactions cannot span over the processor context switches, hence, the transaction can only commit at any point of time within Θ. In TFH, we subdivide Θ into n sub intervals of length τ (i.e., τ_1, τ_2, ..., τ_n), and represent that as a circular buffer B of size n. Any point of the absolute clock time (e.g., ct) can be mapped to an element of B (e.g., $B[\lfloor (ct\%\Theta)/n \rfloor]$). The elements of B, namely *timeline flags*, are set initially to 0, where 0 is the smallest transaction age in our system.

Let the transaction T_i (with age=i) be about to commit, and it needs to preserve ACO. In order to do that, T_i checks the current clock time (ct)[1], and it maps ct to a timeline flag of B (i.e., $B[\lfloor (ct\%\Theta)/n \rfloor]$). If the value of timeline flag equals to i, the transaction commits; then it sets the value of the next M flags after $\lfloor (ct\%\Theta)/n \rfloor$ to $i+1$; recall that B is a circular buffer. Alternatively, if the transaction finds that the value of the timeline flag $\neq i$, then the transaction waits for τ time and perform the check again. However, as the clock time changes, in the next time, it will inspect the next timeline flag.

[1]The *rdtsc* and *rdtscp* instructions load the current value of the processor's timestamp counter, and are safe to be called within hardware transactions.

Interestingly, at any point of time, only one transaction *writes* to a timeline flag that has been never read by any transaction. The reason is that the transaction updates the next M flags, which are corresponding to clock times in the future.

Figure 9.3 illustrates an example of TFH execution with 19 timeline flags. A transaction with successful commit needs to set the next 3 flags (M).

9.3.1 Configuring Timeline Flags

M (window size) represents the number of flags set by the transaction after it commits. The transaction must set enough flags for transferring the ordering information to the next transaction. This means that if the next transaction execution takes more than $M \times \tau$ after the commit of its predecessor transaction, then the ordering information will not be found until the next context switch (e.g., if M=2 in Figure 9.3). Another important configuration is the length of τ, as it represents an upper bound on the commit frequency. A large value of τ will add undesired delays; while small τ values will increase the number of timeline flags (meta-data), consequently, the probability of cache capacity aborts increases.

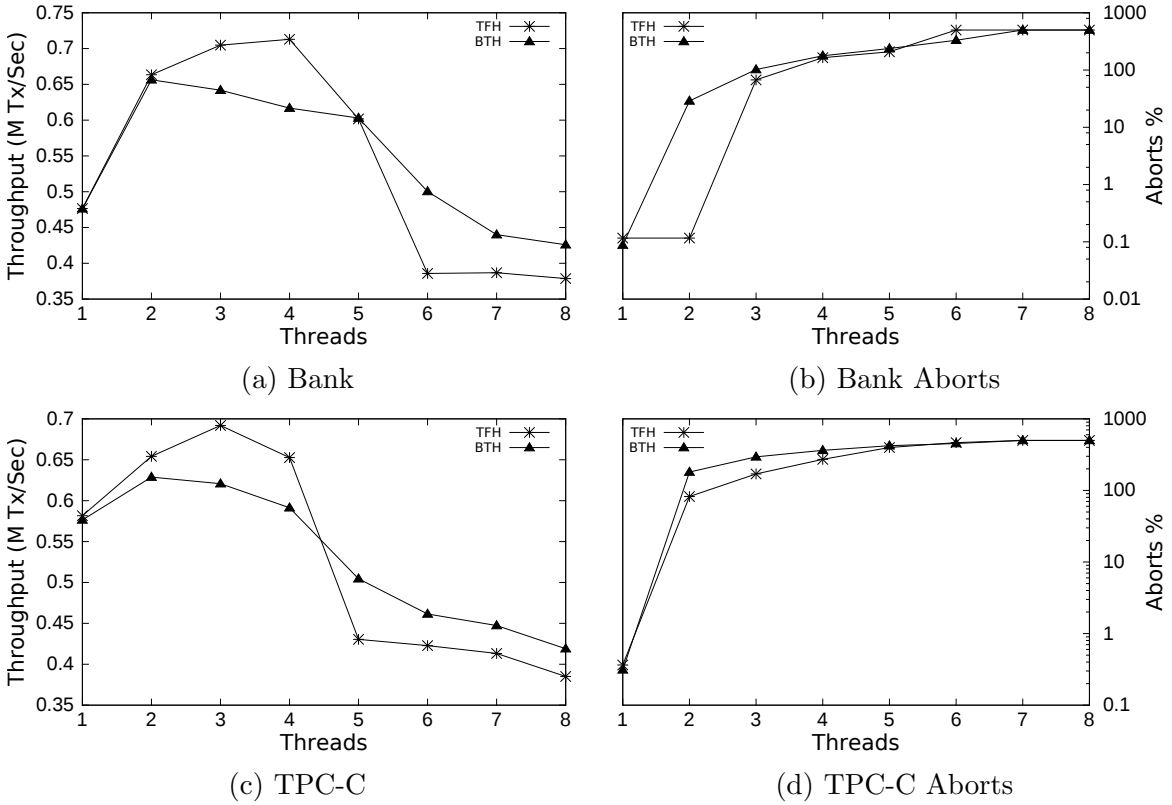
9.3.2 Evaluation

In this section, we compare our two algorithms: BTH and TFH. Both algorithms try to use the fast-path using HTM for 5 times, then fallback to the slow-path using the Single Global Lock (SGL) [189] technique. In BTH, each thread uses 9 tickets distributed over different cache lines. The TFH algorithm uses a circular buffer of 32 flags with a window size (i.e., M) of 10 flags. Each flag covers a clock time period of 256 cycles. The *rdtscp* instruction was used to determine the elapsed time.

To conduct our evaluation, we used two well-known benchmarks: Bank, a micro-benchmark that simulates monetary operations on a set of accounts (see Section 8.6.1), and Transaction Processing Performance Council (TPC-C) [48], an On-Line Transaction Processing (OLTP) benchmark, which simulates an ordering system on different warehouses. Experiments were conducted on a machine equipped with Intel Haswell Core i7-4770 processor with 4 cores, hyper-threading enabled, and the L1 cache size is 8 MB. Results are the average of five runs.

Bank. Figure 9.4a shows the performance of the two algorithms with increasing the number of threads. TFH outperforms BTH with a 1.5× peak performance at 4 threads. Figure 9.4b shows that TFH incurs less aborts than BTH at low number of threads (i.e., 1-4 threads). After 4 threads, the hyber-threading disallow HTM to perform more gain. As a result, the performance of both algorithms degrades due to the aborts and falling back to the slow-path.

TPC-C is an on-line transaction processing (OLTP) benchmark. TPC-C transactions are more complex (i.e., more computation and memory accesses) and longer than the Bank



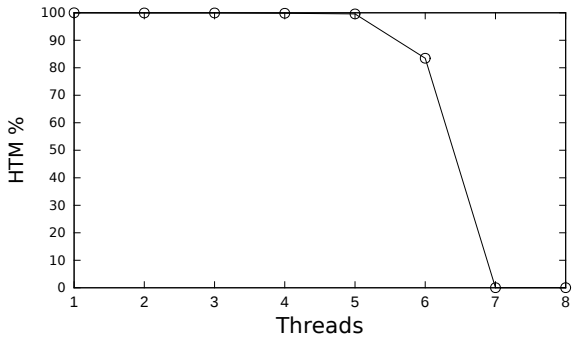
(a) Bank (b) Bank Aborts

(c) TPC-C (d) TPC-C Aborts

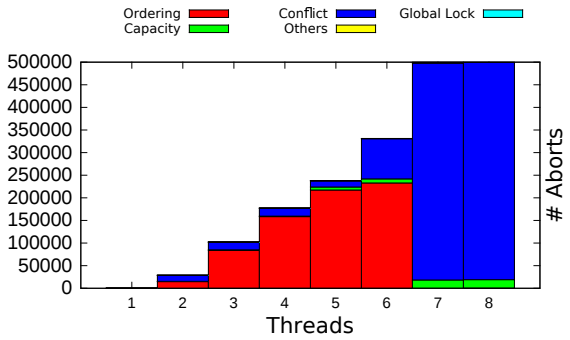
Figure 9.4: BTH vs. TFH using Bank and TPC-C Benchmakrs.

micro-benchmark. TPC-C simulates a set of users executes transactions (entering and delivering orders) against a set of warehouses. Each warehouse (20 warehouses in our configurations) represents a supplier for a set of sales districts (50 districts in our configurations), and each district serves multiple and concurrent customers' requests. Figures 9.4c and 9.4d report the performance and the aborts using TFH and BTH algorithms. TFH outperforms BTH at low number of threads, however, TFH is highly affected by contention at higher number of threads.

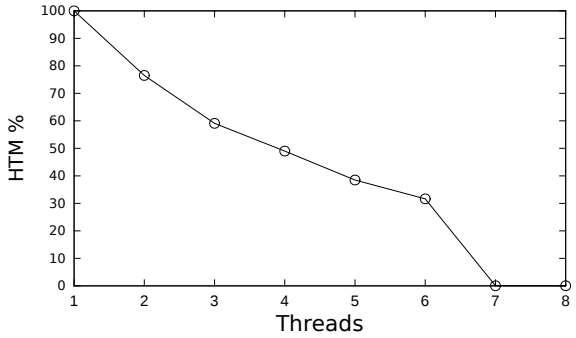
Figure 9.5 illustrates the abort reasons breakdown for each of the benchmarks/algorithms. From the figure we notice that the primary reason for aborts at a low number of threads is the *ordering*, while at higher threads, the *conflict* aborts dominate. TFH incurs less *capacity* aborts than BTH algorithm, thanks to the reduction in the meta-data size; single shared timeline instead of the per-thread tickets. Interestingly, TFH algorithm has a low number of aborts at two threads because the two threads alternate their transaction executions and transfer the ordering information across the timeline flags. This is not the case for the BTH algorithm because the tickets burning technique depends on the transaction length, hence, it is subject to data conflict even with the case of two threads.



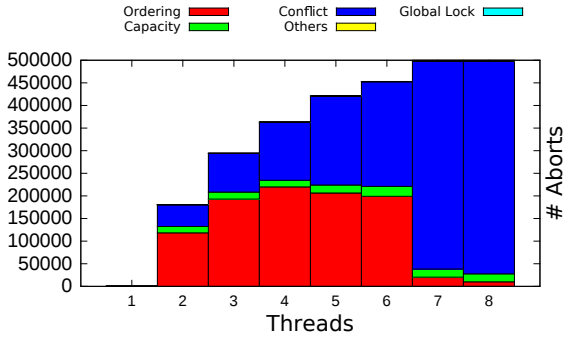
(a) BTH, Bank fast-path to slow-path Ratio



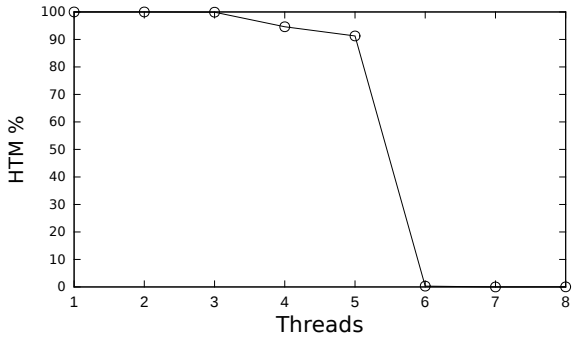
(b) BTH, Bank Aborts Breakdown



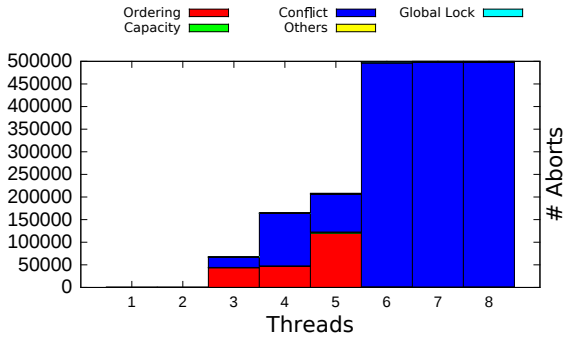
(c) BTH, TPC-C fast-path to slow-path Ratio



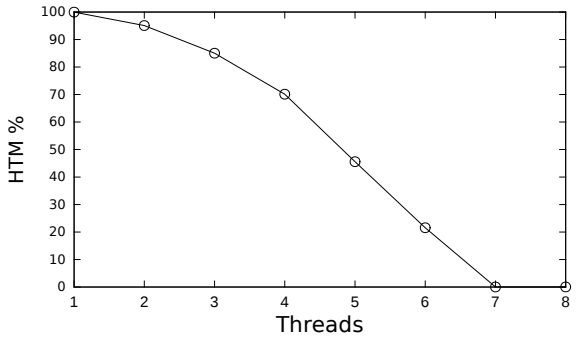
(d) BTH, TPC-C Aborts Breakdown



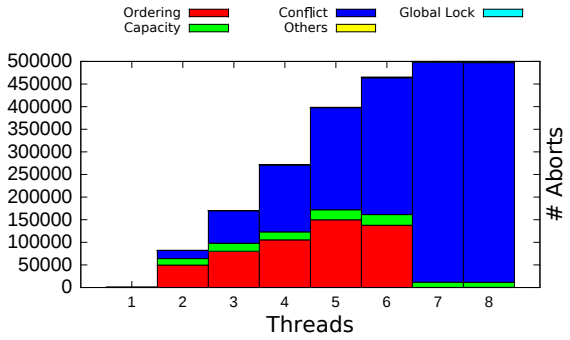
(e) TFH, Bank fast-path to slow-path Ratio



(f) TFH, Bank Aborts Breakdown



(g) TFH, TPC-C fast-path to slow-path Ratio



(h) TFH, TPC-C Aborts Breakdown

Figure 9.5: BTH and TFH fast-path execution

```
transaction {
  x = 1;
  y = 1;
}
```

```
transaction {
  r1 = x;
  r2 = y;
  if(r1 = r2)
      STATEMENT
}
```

Figure 9.6: Execution of two transactions using HyTM

9.4 The Ordered Write Back - Reduced Hardware Algorithm (OWB-RH)

As we discussed in Section 9.1.1, supporting ACO using HTM is challenging and inherently imposes restrictions on the executed code (e.g., capacity and transaction length constraints). In BTH and TFH algorithms we exploit the time to exchange the ordering information between transactions, however, this approach is application and workload dependent and requires fine tuning for the algorithm configurations. In this section, we propose a hybrid TM algorithm; namely OWB-RH. The algorithm extends the OWB algorithm (see Section 6).

9.4.1 Hybrid TM Design Choices

Existing hybrid TM proposals exploits HTM in two ways. The first is by executing the transaction in hardware and falling back to software (slow-path) [36, 52] under certain conditions such as capacity constraints or reaching the threshold of the number of retries. The second usage is by employing the HTM as a tool to enhance the software path. In the later approach, called *reduced hardware (RH)* [119], the slow-path uses a smaller hardware transaction to complete its action. This in effect reduces the hardware fast-path instrumentation and speeds up the slow-path execution.

In the execution shown at Figure 9.6 [52], two concurrent transactions T_1 and T_2 run in hybrid TM system. Algorithms 19 and 20 show inconsistent histories results from the coexistence of two different transactions types (one in software and the other in hardware) running concurrently. In Algorithm 19, the software transaction T_1 writes back its write-set to the memory. Meanwhile, the transaction T_2 (which runs as hardware transaction) observes a partial commit of T_1 (i.e., the value of x) and will execute an invalid code. In Algorithm 20, during the execution of T_2 (which runs as a software transaction) T_1 completes as a pure hardware transaction. As a result, T_2 observes an inconsistent state. Transactions with an inconsistent state may: i) perform actions that are out of control of the TM framework (e.g., invalid memory accesses), ii) execute unexpected control flow such as infinite loops, or iii) do some irrevocable or visible operation. Most hybrid TM proposals take into account such interaction between software and hardware transactions.

From a system perspective, hybrid TM approaches [152, 157] are classified into three cate-

Algorithm 19 A history with inconsistent HTM state.

Initial values are 0

STM_WRITEBACK(T_1)		
x = 1		
	HTM_BEGIN(T_2)	
	r1 = x	
	r2 = y	
	if(r1 = r2)	▷ HTM Inconsistent State
y = 1		

Algorithm 20 A history with inconsistent STM state.

Initial values are 0

	STM_BEGIN(T_2)	
	r1 = x	
HTM_BEGIN(T_1)		
x = 1		
y = 1		
HTM_END(T_1)		
	r2 = y	
	if(r1 = r2)	▷ STM Inconsistent State

gories. The first approach restricts only a single type of transactions to exists at a time (i.e., all software or all hardware) [109]. The drawback of this approach is that once one transaction needs to execute in software, it forces all the system to run in the software slow-path only. The second approach is to permit software and hardware transactions to coexist [52], and preserve correctness and atomicity through a shared metadata. The last approach [157] combines the previous two and allows multiple types of transactions (e.g., pure software, pure hardware, and reduced hardware) to coexist with a phasing option, hence, it is possible to restrict the execution of a certain type of transactions only according to the workload.

9.4.2 Algorithm Description

As a first step in this direction, we propose a hybrid algorithm that exploits HTM to enhance the performance of our OWB algorithm (see Section 6) using the reduced hardware approach. OWB-RH algorithm differs from OWB in the following points:

- The `TryCommit` procedure (see Algorithm 22) uses an inner hardware transaction to write back the changes and acquire the locks atomically. Instead of the use of compare-and-swap operation (line 89 in Algorithm 3), HTM guarantees the required atomicity. However, to reduce the transaction length the write-write conflict (including any cascaded abort) resolution is handled outside the hardware transaction (lines 71-84, Algorithm 22).

- As the write back is done atomically in HTM, there is no need to re-validate the read-set using `Validate_Locked_Reads`.
- The versioned lock was simplified to store the age of the last writer instead of a monotonically increasing counter. This is mainly proposed to reduce the write-set of the hardware transaction, hence reducing the aborts due to the cache capacity. However, as a side effect, it permits reader transactions to stay valid if concurrently a writer transaction modified an address that got aborted afterwards (in OWB this causes a version increment). Furthermore, one of the lock bits ($LOCK\_FLAG$) is reserved for indicating whether it is locked or unlocked. That way, it is possible to access the current (or the last) writer for any address by masking the lock bit. For that reason, the system stores a reference to the last MAX_ACTIVE transactions in a bounded circular array (line 2, Algorithm 21).

The `Read`, `Write`, `Validate_Reads` and `Commit` procedures are similar to the ones in OWB, with the exception of using the age instead the version inside the lock. In `TryCommit`, we need to backup the last writer version before overwriting it, and it is used during the `Abort`.

Finally, The `TryCommit` falls back to the original OWB's `TryCommit` procedure (line 91, Algorithm 22) when a retries threshold is reached, or the transaction is aborted due to cache capacity.

9.4.3 Evaluation

To illustrate the gain of the proposed OWB-RH algorithm, we consider the RSTM micro-benchmarks [2] to evaluate the effect of changing the read-set/write-set sizes on the performance (see Section 5.9.1 for the complete description of the tested applications). As a baseline, we compare against the original OWB algorithm (see Chapter 6).

Experiments were conducted on a machine equipped with Intel Haswell Core i7-4770 processor with hyper-threading enabled (i.e., 4 cores, 8 threads), and the L1 cache size is 8 MB. Results are the average of five runs.

As a general comment on the results, OWB-RH consistently outperforms (or performs the same as) OWB. In RNW1 (see Figures 9.9a and 9.9b), as the write-set is very small (only single address), there is no difference between the two algorithms. However, in RWN and MCAS (see Figures 9.9c-9.9f), OWB-RH benefits from the fast write back using HTM.

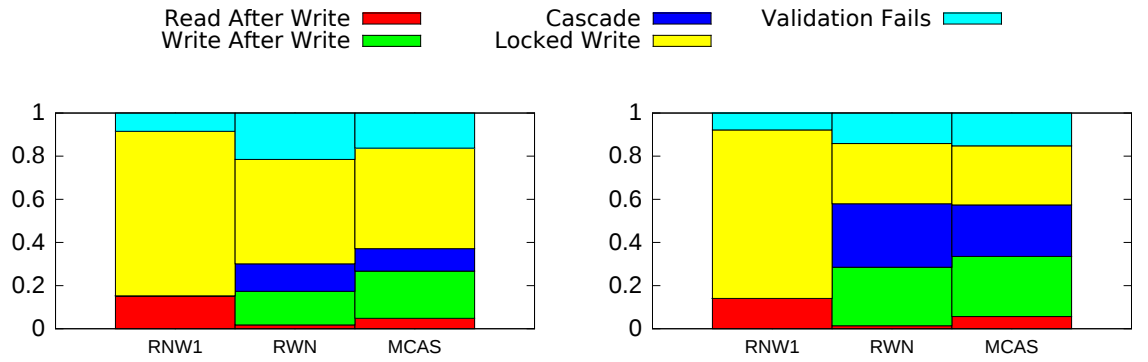
It worth noting that the percentage of aborts reported in Figures 9.9b, 9.9d and 9.9f are the software aborts incurred by OWB-RH. OWB-RH tries to write back its write-set using the HTM (fast-path), and on consecutive aborts, it falls back to the original OWB write back (slow-path). In our OWB-RH implementation, we set the maximum number of fast-path retries to five (see line 70, Algorithm 22). Figure 9.8 shows the details of slow-path execution; to the left (Figure 9.8a) the ratio between transactions that managed to complete

Algorithm 21 OWB-RH - pseudo code

```
1: procedure BEGIN(TRANSACTION tx)
2:     activeWriters[tx.age % MAX_ACTIVE] = tx          ▷ Keep a reference for last active transactions
3:     tx.lastObserved = lastCommitted                  ▷ observe the last committed transaction
4: end procedure

5: procedure READ(SHAREDOBJECT so, TRANSACTION tx)
6:     readVersion = so.version
7:     if tx.writeSet.contains(so) then
8:         tx.readSet.add(so, readVersion)
9:         return tx.writeSet.get(so).value             ▷ Read written value
10:    else if readVersion & LOCK_FLAG then             ▷ Check speculative write
11:        currentWriter = activeWriters[readVersion & LOCK_MASK % MAX_ACTIVE ]
12:        if currentWriter.age > tx.age then
13:            ABORT(currentWriter)                     ▷ W2 → R1; Read after Speculative Write
14:            go to 6
15:        else                                         ▷ W1 → R2; Add Tx to its dependencies
16:            currentWriter.dependencies.add(tx)
17:            if currentWriter.status ≠ ACTIVE then    ▷ Double check writer
18:                ABORT(tx)                            ▷ Writer got aborted while registeration
19:            end if
20:        end if
21:    end if
22:    if readVersion ≠ so.version then
23:        go to 6
24:    end if
25:    Validate_Reads(tx)
26:    tx.readSet.add(so, readVersion)
27:    return so.value
28: end procedure

29: procedure WRITE(SHAREDOBJECT so, OBJECT value, TRANSACTION tx)
30:    tx.writeSet.add(so, newValue)                    ▷ Save new value
31: end procedure

32: procedure VALIDATE_READS(TRANSACTION tx)
33:    if tx.lastObserved ≠ lastCommitted then
34:        tx.lastObserved = lastCommitted
35:        for each Entry entry in tx.readSet do        ▷ Validate Read Set
36:            SharedObject so = entry.so
37:            if so.version ≠ entry.readVersion then
38:                return ABORT(tx)                     ▷ Read a wrong version
39:            end if
40:        end for
41:    end if
42:    return VALID
43: end procedure

44: procedure ABORT(TRANSACTION tx)
45:    if tx.status = ABORTED then return false; end if   ▷ Already got aborted
46:    if tx.status = INACTIVE then return false; end if  ▷ Already compeleted
47:    while ! CAS(tx.status, ACTIVE, TRANSIENT) do       ▷ Try Abort
48:        repeat until tx.status ≠ TRANSIENT             ▷ Spin Wait
49:        go to 45
50:    end while
51:    for each Transaction dependency in tx.dependencies do
52:        ABORT(dependency)                            ▷ Abort dependent transactions
53:    end for
54:    for each Entry entry in tx.writeSet do
55:        SharedObject so = entry.so
56:        if so.version = (tx.age | LOCK_FLAG) then    ▷ Aquired lock
57:            so.value = entry.newValue                ▷ Revert value
58:            so.version = entry.version               ▷ Revert version
59:        end if
60:    end for
61:    tx.status = ABORTED
62:    return true
63: end procedure
```

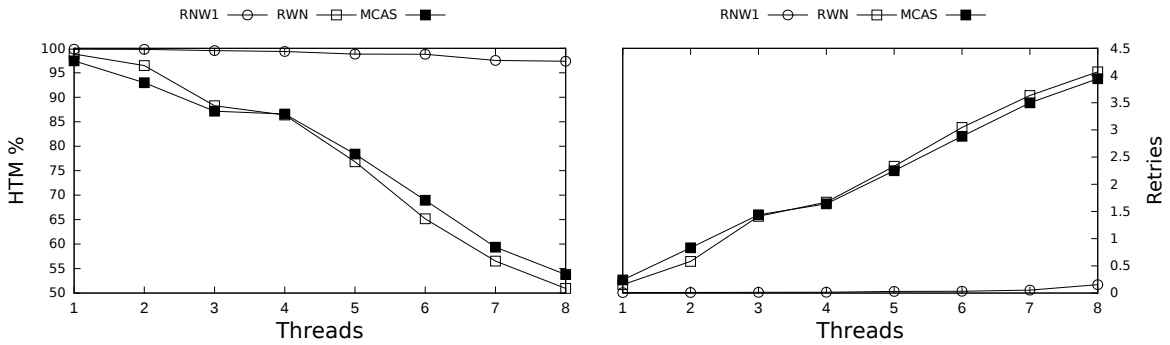
Algorithm 22 OWB-RH - Commit.

```
63: procedure TryCommit(Transaction tx)
64:     if tx.status = ABORTED then return false; end if                    ▷ Already got aborted
65:     while ! CAS(tx.status, ACTIVE, TRANSIENT) do                        ▷ Try Commit
66:         repeat until tx.status ≠ TRANSIENT                              ▷ Spin Wait
67:         return false
68:     end while
69:     if VALIDATE_READS(tx) ≠ VALID then return false; end if
70:     retries = MAX_RETRIES
71:     for each Entry entry in tx.writeSet do                              ▷ Lock Write Set
72:         SharedObject so = entry.so
73:         currentVersion = so.version;
74:         if currentVersion & LOCK_FLAG then
75:             writerAge = currentVersion & LOCK_MASK
76:             if tx.age < writerAge then
77:                 currentWriter = activeWriters[writerAge % MAX_ACTIVE ]
78:                 ABORT(currentWriter)                                    ▷ W2 → W1; Write after Specu. Write
79:             else
80:                 ABORT(tx)                                               ▷ W1 → W2; Write after Write
81:                 return false
82:             end if
83:         end if
84:     end for
85:     status = XBEGIN                          ▷ ——————————————————— HTM Start
86:     if status = _XABORT_EXPLICIT then
87:         go to 71                                                        ▷ Resolve Write-Write conflict
88:     else if status ≠ _XBEGIN_STARTED then
89:         retries - -                                                     ▷ Count HTM fail
90:         if retries = 0 ∨ status = _XABORT_CAPACITY then
91:             return owb::TryCommit(Tx)                                   ▷ Fallback to software slow-path
92:         end if
93:         go to 85                                                        ▷ Retry
94:     end if
95:     myLock = tx.age | LOCK_FLAG
96:     for each Entry entry in tx.writeSet do
97:         SharedObject so = entry.so
98:         if so.version & LOCKED_FLAG ∧ so.version ≠ myLock then          ▷ Check Write-Write Conflict
99:             XABORT                                                      ▷ Abort to Resolve Write-Write Conflict
100:        end if
101:        entry.version = so.version                                      ▷ Save old version
102:        so.version = myLock                                             ▷ Lock the address
103:        temp = so.value                                                 ▷ Save old value
104:        so.value = entry.newValue                                       ▷ Expose written value
105:        entry.newValue = temp
106:    end for
107:    XEND                                 ▷ ——————————————————— HTM Commit
108:    tx.status = ACTIVE                                                  ▷ Transaction Exposed
109:    return true
110: end procedure

111: procedure Commit(Transaction tx)
112:    if tx.status = ABORTED then return false; end if                    ▷ Already got aborted
113:    while ! CAS(tx.status, ACTIVE, TRANSIENT) do                        ▷ Try Complete
114:        repeat until tx.status ≠ TRANSIENT                              ▷ Spin Wait
115:        return false
116:    end while
117:    if VALIDATE_READS(tx) ≠ VALID then return false; end if
118:    for each Entry entry in tx.writeSet do
119:        SharedObject so = entry.so
120:        so.version = tx.age                                             ▷ Unlock
121:    end for
122:    tx.status = INACTIVE                                                ▷ Transaction Committed
123:    return true
124: end procedure
```



(a) OWB Algorithm (b) OWB-RH Algorithm

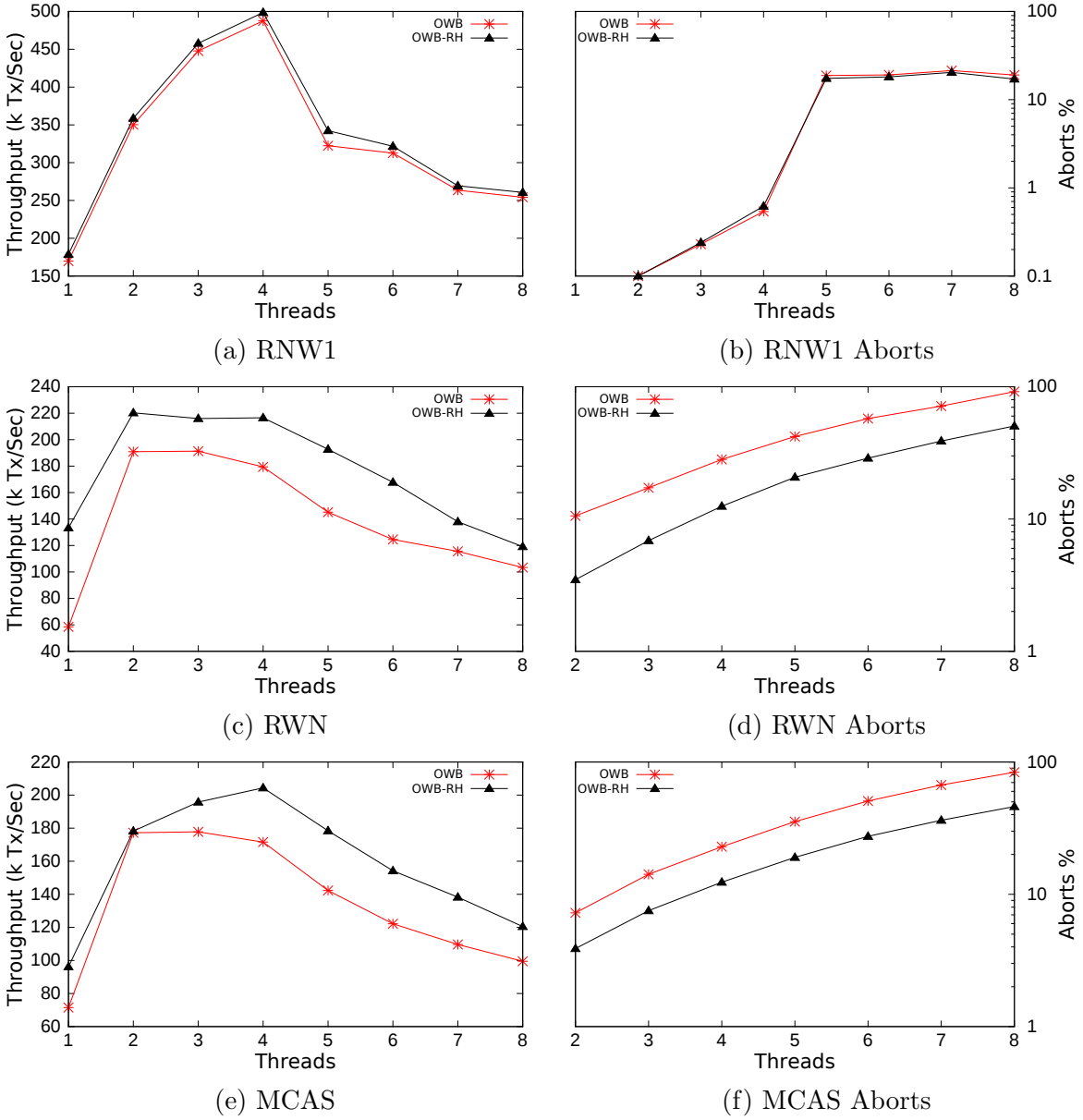
Figure 9.7: Aborts Breakdown



(a) Fast-path to slow-path Ratio (b) Average number of fast-path retries per transaction

Figure 9.8: OWB-RH fast-path execution

its execution using only the fast-path and the transactions that fall back to the slow-path, while Figure 9.8b depicts the average number of slow-path retries per transaction. In RNW1 benchmark, with rare conflicts, the transaction always succeed to use the fast-path. However, in RWN and MCAS, transactions incurs repeated conflicts and fall back to the slow-path. Interestingly, after 4 threads, the percentage of transactions that fails to complete using slow-path increases notably, this is due to the hyper-threading effect. Figures 9.9b, 9.9d and 9.9f shows a reduction of the number of aborts in OWB-RH, however, such reduction is due to reducing the *validation* and *locked write* aborts (see Figure 9.7); both of these aborts types incurred mostly during the commit. For the *locked write* aborts, as the fast-path transaction retries multiple times, some of these aborts are resolved. In other words, the slow-path retries have a useful back-off effect on the commit execution. On the other hand, the slight reduction of the *validation* aborts is due to the use of age instead the monotone counter.



(a) RNW1 (b) RNW1 Aborts

(c) RWN (d) RWN Aborts

(e) MCAS (f) MCAS Aborts

Figure 9.9: OWB vs. OWB-RH using RSTM Micro-benchmark.

Chapter 10

Conclusions and Future Work

The vast majority of applications and algorithms are not designed to exploit properly the new trend of multi-processor chips design, which creates a gap between the available commodity hardware and the running software. This gap is expected to continue for years with the burdens of developing and maintaining parallel programs. In this dissertation, we aim at extracting coarse-grained parallelization from sequential code. We exploited transactional memory (TM) as an optimistic concurrency technique for supporting safe memory access and introduced algorithmic modifications for preserving program chronological order. TM is known with its execution overhead, which could be comparable to locking overhead, but in comparison to sequential code it could outweigh any performance gain. In this thesis we tackled this issue in several ways such as: employing static analysis, fast-path sequential execution, transactional pooling, transactional increments, prioritize transactions, exploiting low-level semantics, and getting use of HTM support.

Summarizing, in this thesis we present the following solutions.

We presented HydraVM, a JVM that automatically refactors concurrency in Java programs at the bytecode level. Our basic idea is to reconstruct the code in a way that exhibits data-level and execution-flow parallelism. STM was exploited as memory guards that preserve consistency and program order. Our experiments show that HydraVM achieves speedup between 2×-5× on a set of benchmark applications.

We presented Lerna, a completely automated system that combines a software tool and a runtime library to extract parallelism from sequential applications without any programmer intervention. Lerna leverages software transactions to solve conflicts due to data sharing of the produced parallel code, thus preserving the original application semantics. Using Lerna is finally possible ignoring the application logic and exploiting the cheap hardware parallelism using a blind parallelization. Lerna showed great results with multiple benchmarks with average 2.7× speedup over the original code.

Both frameworks propose novel techniques over past STM-based parallelization works in that

they benefit from static analysis for reducing transactional overheads, permit accessing some reducible variables allocated on the stack (not just global variables as [120]), target arbitrary programs (not just recursive ones as [32] does), is entirely software-based (unlike [32, 66, 56, 182]), do not require program source code.

With OWB, OWB-RH, OUL, and OUL-steal algorithms we effectively address the problem of committing transactions with an order defined prior to execution. We show that even if a system requires a specific commit order, it is possible to achieve high performance exploiting parallelism with data conflicts. Results show important trends: our OUL outperforms other ordered competitors consistently. In particular, the maximum speedup achieved is 4× over Ordered TL2, 4.3× over Ordered NORec, 8× over Ordered UndoLog visible, 10× over Ordered UndoLog invisible, and 5.7× over STMLite [120]. Interestingly, the peak gain over the sequential non-instrumented execution in micro benchmarks is 10× and 16.5× in STAMP.

With the proposed new TM extensions, we show that it is possible exploiting application semantics in a transparent manner, without involving the programmer. We did so by identifying *TM-friendly* semantics and proposing an approach to inject them into the current TM algorithms and frameworks. We also integrated our work in GCC and provide a full compiler support for them. Our experimental results depicted a promising improvement over the base algorithms. A possible extension of this line of research is investigating more on including HTM algorithms and supporting more complex semantic patterns.

10.1 Discussion & Limitations of Parallelization using TM

TM provides mechanisms to handle data dependencies at run-time, but comes with an overhead when TM is used in parallelization of sequential code. The primary factors of overhead are: synchronizing shared accesses, preserving chronological order at commit, read-set validation, and aborting transactions because of actual data dependencies, control flow dependencies, or false conflicts. Some code patterns are TM-parallelization friendly such as: code with local computations and few shared accesses; long iteration with shared accesses at its end; workload with balanced computation access iterations; and code patterns that accesses flat data structures (i.e., those with no single point of contention).

On the other hand, the performance of the TM parallel code is highly dependent on the ability to exclude addresses, which are not needed to be accessed (e.g., read-only addresses). This could be done using an alias analysis or by a high-level user intervention. Parallelization using TM becomes less effective when:

- there are loops with few iterations because the actual application parallelization degree is limited;

- there is an irreducible global access at the beginning of each loop iteration, thus increasing the chance of invalidating most transactions from the very beginning;
- workload is heavily unbalanced across iterations and with frequent control-flow changes;
- extensive usage of multiple levels of pointers, which hinders the use of alias analysis and force using transactional access even for addresses calculations; and
- when the size of the shared read accesses increases significantly, consequently, the validation overhead increases.

Due to the above limitations, one of the main factors that affect the efficiency of parallelization using TM is the ability to move the patterns that are not suited to be parallelized using TM outside the parallel TM blocks. We did that by exploiting static analysis techniques. However, static analysis may not be sufficient to detect complex patterns. In Section 10.1.1, we show some of the recommended programming practices to limit or avoid these complex (irreducible) patterns.

Finally, two features distinguish TM over other parallelization techniques: its ability to get use of the application (workload) semantics to increase the concurrency, even with the existence of true conflicts; and the hardware support of the mainstream commodity processors, such as Intel's Haswell.

10.1.1 Recommended Programming Patterns

In our work, we assume that there is no user intervention. However, we identify here some programming patterns that would help our work to perform better. The key points in these guidelines are: 1) helping the alias analysis to detect some relations between the accessed locations, which helps in reducing the number of the transactional accesses and exploiting our semantic APIs; and 2) enhancing the transactional execution by minimizing the cost of retries or reducing the transaction length.

Avoid Unneeded Global Variables

The excessive use of global variables are generally undesirable because it leads the program to have: an unpredictable global state of , poor testability, and less code comprehension. In the context of parallelization, in order to guarantee memory consistency each access to a global variable needs to be monitored. This enforces the alias analysis technique to do more work for checking this access, and it may end with the *may-alias* conclusion, which enforces using transactional access. Recall that the transactional access is costly as it enlarges the transaction metadata and slows down the validation process.

Use Simple Loops

`Foreach` style of loops is commonly used instead of the standard loops (e.g., `for` with counter, or the conditional `while`). `Foreach` simplifies the programming by declaring some logic to be applied on each element in a collection instead of doing this logic n times. However, in our work the `Foreach` loops are highly undesirable because it maintains an internal state (e.g., iterator) which builds an interdependency between the iterations. This internal state cannot be analyzed or reduced simply, thereby the loop may not be parallelized.

Relocate Local Processing

As TM employs the "rollback-and-retry" mechanism for the aborted transactions, each time a transaction is aborted it will need to re-execute all the computation within the transaction boundaries. These computations may be on local or shared variables. If the computations are local, then we highly recommend that they are moved before the first shared access. That way we can push the transaction start till after these computation, and hence we can reduce the transaction length and the retry cost. If this is not possible, then it would be beneficial to move the computations to the end of the transaction after accessing all global reads. Thus, if the transaction conflicts with another one, it will detect that at earlier stage of its life, so the retry will be less costly.

Avoid Pointer Arithmetic

Performing pointer arithmetic is usually unsafe because it may lead to invalid accesses to memory. Additionally, with arithmetic pointer operations the automatic parallelization has nothing to do except accessing it transactionally. This conservative action may lead to a considerable runtime overhead. We encourage substituting the pointer arithmetic, if possible, with simple base/offset accesses.

Use Pure Functions

The *"pure"* functions are the ones whose result depend only on the input parameters and have no side effects (i.e., affect the global program state). Organising the code to exploit pure functions, whenever possible, allows us to reason about the semantics of these pure functions. In most cases, pure functions can be identified as "safe" functions, so we will not need to include them into the transactional access or the alias analysis. This has a good impact on reducing both the compilation time and the runtime overhead.

No Functions Pointer, Use Templates!

Pointer to functions are used to generalize access to program logic or to provide a callback function. We identified a wide usage of pointer to function in STAMP [37] and PARSEC [133] (e.g., comparator functions). However, in many cases this generality is not essential. For example, a program that uses Kmeans algorithm with data points stored as *floats* will not decide dynamically at runtime to use Kmeans with *integer* data points. If Kmeans code extracts the data points *compare* logic as a pointer to function to provide generality, then it will add a big limitation on the parallelization. Even with TM, it is not possible to handle a code with a pointer to function, because such code may be irrevocable. An alternative approach is the use of *code templates*. Code templates (or *generics* in Java) permit the programmer to write a generic code that is interpreted at compile time. That way, the code will not lose its abstraction and generality and the parallelization framework will be able to optimize it.

Data Structures Accesses

PARSEC [133] is an example of a benchmark that uses data structures extensively. As we cannot inline each call to the data structure operations (e.g., `add`, `remove` or `contains`), these operations are instrumented and executed transactionally. As data structure operations are called frequently, we faced a considerable overhead with these applications. Thus, our recommendation is to decouple the parallelization of the application from its underlying data structure, and to replace these data structures with concurrent thread-safe versions. That way, data structure operations will be marked as *safe* calls, and the parallelization effort will focus on the application logic.

Immutable Objects

Objects with immutable state (e.g., immutable singleton) are thread safe because they do not permit changes to their state. Refactoring the code to use immutable objects and identifying these objects will help the parallelization framework significantly. Examples of immutable objects are: string utility, services registry, or mathematics engine.

Irrevocable Operations Handling

One of the TM limitations is the inability to handle irrevocable operations (e.g., I/O). A loop with many computations and a single irrevocable operation (e.g., read input from file, or print to screen) is not a candidate for parallelization using TM. One possible solution is to split the loop into two parts, and buffer the irrevocable action (e.g., store the read value

or the value to print) in memory. In this case, one of the loops will be irrevocable (i.e., the one that buffer the I/O), while the other loop with the computations will be parallelizable.

10.2 Future Work

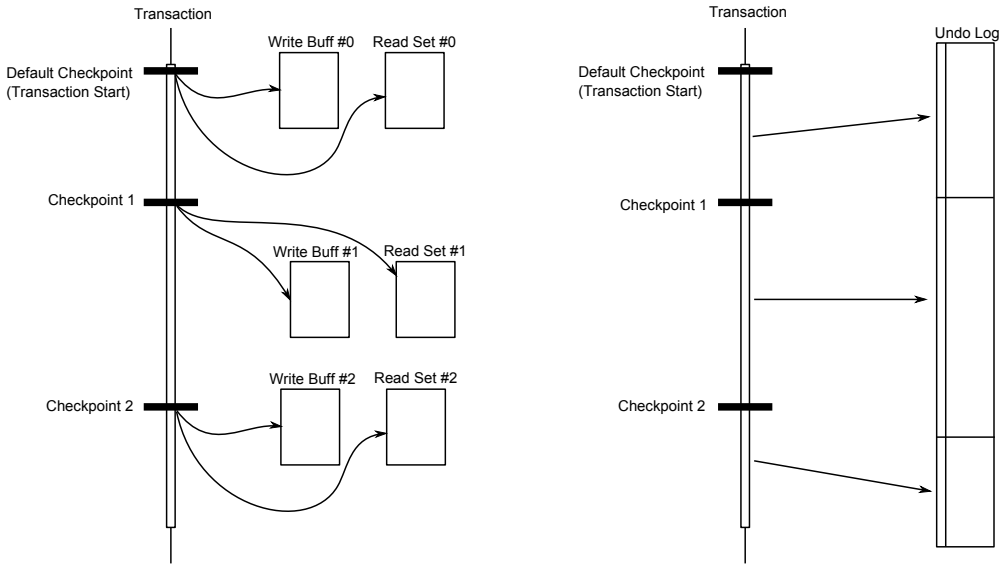
10.2.1 Transaction Checkpointing

Aborting a transaction is a costly operation, not only because the rollback cost, but retrying the code execution doubles the cost and wastes precious processing cycles. Additionally, transactions may do a lot of local processing work that does not use any protected shared variables. A possible technique is *transaction checkpointing*, which creates multiple checkpoints at some execution points. Using checkpoints, a transaction saves the state of the work done at certain times after doing a considerable amount of work. Later, if the transaction experienced a conflict due to an inconsistent read or because writing a value that should be read by another transaction executing an earlier chronological order code, then it can return to the last state wherein the execution was valid (i.e., before doing the invalid memory operation). Checkpointing introduces an overhead for creating the checkpoints (e.g., saving the current state), and for restoring the state (upon abort). Unless the work done by the transaction is large enough to outweigh this overhead, this technique is not recommended. Also, checkpointing should be employed when the abort rate exceeds a predefined threshold. A good example that would get use of this technique is when a transaction finishes its processing by updating a shared data structure with a single point of insertion (e.g., linked list, queue, stack); Two concurrent transactions will conflict when trying to use this shared data structure. With checkpointing, the aborted transaction can jump back till before the last valid step and retry the shared access step.

Transactions are defined as a unit of work; *all or none*. This assumption mandates executing the whole transaction body upon conflicts, even if the transaction has executed correctly for a subset of its lifetime. In the context of parallelization, restarting transactions could prevent any possible speedup; especially with preserving order and executing balanced transactions with equal processing time (e.g., similar loop iterations).

With transaction checkpointing, a conflicting transaction can select which is the best checkpoint wherein the execution was valid and retry execution from it. Inserting checkpoints can be done using different techniques:

- *Static Checkpointing*. Insert new checkpoint at equal portions of the transaction. This is done statically at the code.
- *Dependency Checkpointing*. Alias analysis and memory dependency analysis provides a best-effort guess for memory addresses aliasing. A common situation is the *may alias* result, which indicates a possible conflict. Placing a checkpoint before *may alias*



(a) Write-buffer (b) Undo Log

Figure 10.1: Checkpointing implementation with write-buffer and undo-logs

accesses could improve the performance; a transaction will continue execution if *no alias*, and will retry execution right before the invalid access at *exact alias* situations.

- *Periodic Checkpointing.* During runtime, we checkpoint the work done at certain times (e.g., after doing a considerable amount of work).
- *Last Conflicting Address.* In our model, transactions executing symmetric code (i.e., iteration). Conflicting transactions usually continue conflicting at successive retrials. A conflicting address represents a guess for possible transactions collisions, and could be used as a hint for placing a checkpoint (i.e., before accessing this address).

The performance gain from using any of these proposed techniques is application and workload dependent. Additionally, checkpointing introduces an overhead for handling the checkpoints; unless the work done by the transaction is large enough to outweigh this overhead, it is not recommended. Checkpointing is useful when a transaction finishes its processing by updating a shared data structure (e.g., Fluidanimate see Section 5.9.3); with checkpointing, the aborted transaction can jump back till the checkpoint before the shared access and retry execution.

In order to support checkpointing a transaction must be *partially* aborted. As mentioned before, TM algorithms use either eager or lazy versioning through the usage of a *write-buffer* or an *undo log*. For eager TM, the undo log need to store a meta-data about the checkpoint (i.e., when it occurs). Whenever transaction wants to partially rollback, the undo-log is used to undo changes until the last recorded checkpoint in the log (See Figure 10.1b). With

lazy TM implementations, the write-buffer must be split according to the checkpoints. Each checkpoint is associated with a separate write-buffer stores its changes (See Figure 10.1a). Upon conflict, transaction discard write-buffers for checkpoints exist after the conflicting location. Drawbacks of this approach are: read operations needs to check multiple write-sets, and write-buffers should not be overlapped. Another alternative approach is to consider check checkpoint as a nested transaction, and employs closed-nesting techniques for handling partial rollback. However, supporting closed-nesting introduces a considerable overhead to the TM performance and complicates the system design.

10.2.2 Complex Semantic Expressions

An interesting feature of the semantic operations listed in Table 8.1 is that they can compose by having more than one operator and/or more than one variable in the conditional expression. For example, the scenario shown in Algorithm 9 can be further enhanced if we consider the whole conditional expression (i.e., `TM_READ(x) > 0 || TM_READ(y) > 0`) as one semantic read operation. In this example, if the condition was initially `true` and then a concurrent transaction modifies only one variable, either x or y, to be negative, considering the clause as a whole avoids aborting T_1 given the `OR` operator. A similar enhancement consists of allowing complex expressions in conditional statements (e.g., `x + y > 0`), where modifications on multiple variables may compensate each other so that the return value of the overall expression remains unchanged.

Compositional and complex expressions can be generalized, in the same way as presented in Section 8.2, as an abstract method `cmp(address1, address2, ..., val1, val2, ...)`, where the number of arguments depends on the expression. This way, S-NOrec and S-TL2 can handle them similar to any other `cmp` operation, as long as they guarantee that all the included variables inside an expression are read consistently.

On the other hand, integrating such semantics in the compiler passes is more complicated because it requires to first identify the code pattern that matches the conditional expression. Unfortunately, unlike the "trivial" expressions mentioned in Table 8.1, those complex expressions are transformed by the compiler into different basic blocks. Thus, handling those conditions as a single semantic read operation during the compilation process requires an inter-block analysis. From a compiler design perspective, this optimization will add a non-negligible overhead to the compilation process. This difficulty raises a compromise on the practicality of supporting them at the compiler level, especially if the trivial expressions already cover the common use cases at the application level. At the current stage, we handle compound conditions as multiple semantic reads. A deeper investigation on those expressions is needed in order to know the best way to solve the tradeoff between saving additional aborts and complicating the compiler passes.

10.2.3 Semantic-Based HTM

The challenges of injecting semantics into STM algorithms and HTM algorithms are very different. Concurrency controls in STM are entirely performed and integrated into software frameworks. That allows any sort of modification to the transaction execution, including embedding our extension of defining new semantic constructs and calling them instead of the classical ones (i.e., TM_READ and TM_WRITE). On the other hand, the current HTM release [149, 35] leverages hardware for detecting conflicting executions and gives very limited chances for optimization to the TM framework. For instance, it leaves no control on modifying the granularity of the speculation in HTM transaction; in other words, every memory access within the boundaries of an HTM transaction is monitored by the hardware itself (exploiting an enhanced cache coherency protocol). As a result, executing HTM transactions means preventing any straightforward solution for replacing the basic TM_READ/TM_WRITE constructs with semantic calls (as it can be done for STM).

Although injecting semantics in HTM algorithms is harder than STM, we believe that there is still room for that. For example, the following two approaches can be adopted separately:

- Injecting semantics in the software fallback path of the HTM transaction, similar to how we injected them in pure STM algorithms. For example, RH-NOrec [119] is an HTM algorithm whose software fallback paths are similar to NOrec and can be enhanced similar to S-NOrec.
- Exploiting our compilation passes to make further enhancements. For example, if the conflicting reads/writes are shifted by the compiler to the end of the transaction, the probability of raising a conflict at runtime decreases. Compilation-time solutions do not require modifying the execution pattern of HTM transactions at runtime (which is impossible in current HTM models), providing a good direction to overcome the limited flexibility of HTM APIs.

Regarding the first approach, we investigated the possible alternatives for the software fallback path published in literature so far. Interestingly, we found that most of them are compliant with our former proposals on STM. For example, NOrec has been considered in literature [52, 152, 119] as one of the best STM candidates to integrate with HTM transactions because it has only one global lock as shared metadata, which minimizes the overhead of monitoring the metadata inside HTM transactions (recall any speculation on the meta-data affects the number of cache lines occupied and may generate false conflicts). Among those proposals, RH-NOrec [119] is known to be one of the best-performing HTM algorithms. The main idea of RH-NOrec is to execute transactions in one of three modes: *fast-path*, in which a transaction is entirely executed in HTM; *slow-path*, in which the body of a transaction is executed in software similar to NOrec and the commit phase is executed using a small HTM transaction; and *slow-slow-path*, which is NOrec itself. These three modes are made consistent by speculating the global lock in the HTM transactions. The semantics support

can be injected in the *slow-path* and the *slow-slow-path* of RH-NOrec in a similar way to what we did for NOrec.

The second approach for injecting semantics into HTM algorithms is to involve the compiler. Compilation-time solutions do not require modifying the execution pattern of HTM transactions at runtime (which is impossible in current HTM models), providing a good way of overcoming the limitations of HTM APIs.

The order of executing reads/writes inside transactions is one of the main reasons for raising/avoiding conflicts at runtime. For example, if the conflicting reads/writes are shifted by the compiler to the end of the transaction, the probability of raising a conflict at runtime is minimized. A further optimization on the compilation-level semantic operations defined in Table 8.1 is by reordering them at compilation time (if possible). As a preliminary example, the increment/decrement operations can be delayed to the end of the transaction if the incremented/decremented variables are not read later in the transaction. Also, conditional branches can be "optimistically" calculated before a transaction starts, resulting in a deterministic branch selection inside that transaction, and then the optimistic branch selection can be revalidated at the end of the transaction, to preserve serializability [20]. The proposed compilation-time optimizations are valid for both STM and HTM, although HTM benefits more from that because of its limited APIs.

An appealing research direction is to investigate the different correctness guarantees that can be provided by altering operations at compilation time, such as opacity [76], serializability [20], and publication/privatization safety [121, 117]. Regarding the former examples, optimistic brach selection clearly breaks opacity, similar to the lazy subscription issue discussed in [57]. Also, shifting increments/decrements may break publication-safety in some cases, as discussed in [156]. Making an investigation on the *theoretical* and *practical* possibilities/impossibilities in that direction is one of the objectives of our future work.

10.2.4 Studying the Impact of Data Access Patterns

Transactional Memory does not impose assumptions on the data access patterns as it tolerates data conflicts. In our work, we tried to infer the data access patterns from the conflict rates and adapted the execution according to that (Lerna's adaptive runtime and HydraVM's adaptive online architecture). For example, in *matrix multiplication* if the calculations of each element are assigned to a separate transaction executing in different threads, then each consecutive N (e.g., two) transactions will conflict on updating the adjacent memory locations. The same behavior appears in the membership of *Kmeans* data points calculated by concurrent transactions.

In some situations, it is possible for the user to provide hints about the data access pattern of the applications. In this case, the parallelization framework will not need to "learn" from the execution (e.g., profiling or adaptive execution), and instead will exploit these hints for

better assignments of data to transactions (threads). Furthermore, knowing the nature (e.g., size, time or reference locality) of the processed data in advance allows the parallelization framework to select the best configurations for the underlying architecture (e.g., cache lines, thread stack size).

Bibliography

[1] Intel Parallel Studio. https://software.intel.com/en-us/intel-parallel-studio-xe.

[2] RSTM: The University of Rochester STM. http://www.cs.rochester.edu/research/synchronization/rstm/.

[3] The LLVM Compiler Infrastructure. http://llvm.org.

[4] TinySTM: A time-based STM. http://tinystm.org/tinystm.

[5] Intel transactional memory compiler and runtime application binary interface. https://software.intel.com/sites/default/files/m/5/a/2/a/f/8097-Intel_TM_ABI_1_0_1.pdf, 2008.

[6] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 26–37, Jun 2006.

[7] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft specification of transactional language constructs for c++, 2009.

[8] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, pages 65–79, 2011.

[9] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting postdominance for speculative parallelization. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 295–305. IEEE, 2007.

[10] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[11] T. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6 –16, Jan. 1990.

[12] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM.

[13] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. Safety of live Transactions in Transactional Memory: TMS is necessary and sufficient. In *DISC*, pages 376–390, 2014.

[14] H. Avni and B. C. Kuszmaul. Improving HTM scaling with consistency-oblivious programming. In *9th Workshop on Transactional Computing*, TRANSACT '14, 2014. Available: `http://transact2014.cse.lehigh.edu/`.

[15] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *High Performance Computing–HiPC 2005*, pages 465–476. Springer, 2005.

[16] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[17] U. Banerjee. *Dependence analysis*, volume 3. Springer Science & Business Media, 1997.

[18] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui. Unifying thread-level speculation and transactional memory. In *Proceedings of the 13th International Middleware Conference*, pages 187–207. Springer-Verlag New York, Inc., 2012.

[19] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *In Proceedings of the 35th 8 International Symposium on Computer Architecture*, 2008.

[20] P. A. Bernstein and N. Goodman. Serializability theory for replicated databases. *J. Comput. Syst. Sci.*, 31(3):355–374, 1985.

[21] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[22] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 99–108, New York, NY, USA, 2002. ACM.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[24] G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, 2003.

[25] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[26] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.

[27] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[28] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[29] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[30] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Parallel Comput.*, 24:421–444, May 1998.

[31] B. Bradel and T. Abdelrahman. Automatic trace-based parallelization of java programs. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, page 26, sept. 2007.

[32] B. J. Bradel and T. S. Abdelrahman. The use of hardware transactional memory for the trace-based parallelization of recursive java programs. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 101–110, New York, NY, USA, 2009. ACM.

[33] R. L. H. C. C. M. Bratin Saha, Ali-Reza Adl-Tabatabai and B. Hertzberg. McRT-STM: a high performance software transactional memorysystem for a multi-core runtime. In *PPOPP*, pages 187–197, 2006.

[34] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society.

[35] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. 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.

[36] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *9th Workshop on Transactional Computing*, TRANSACT '14, 2014. Available: `http://transact2014.cse.lehigh.edu/`.

[37] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[38] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[39] B. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Executing Java programs with transactional memory. *Science of Computer Programming*, 63(2):111–129, 2006.

[40] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. *ACM SIGPLAN Notices*, 41(6):1–13, 2006.

[41] B. Chan. The umt benchmark code. *Lawrence Livermore National Laboratory, Livermore, CA*, 2002.

[42] B. Chan and T. Abdelrahman. Run-time support for the automatic parallelization of Java programs. *The Journal of Supercomputing*, 28(1):91–117, 2004.

[43] M. Chen and K. Olukotun. Test: a tracer for extracting speculative threads. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 301–312. IEEE, 2003.

[44] M. K. Chen and K. Olukotun. The jrpm system for dynamically parallelizing java programs. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 434–445. IEEE, 2003.

[45] P. Chen, M. Hung, Y. Hwang, R. Ju, and J. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *ACM SIGPLAN Notices*, volume 38, pages 25–36. ACM, 2003.

[46] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. *ACM SIGPLAN Notices*, 34(10):1–19, 1999.

[47] R. A. Chowdhury, P. Djeu, B. Cahoon, J. H. Burrill, and K. S. McKinley. The limits of alias analysis for scalar optimizations. In *Compiler Construction*, pages 24–38. Springer, 2004.

[48] T. Council. tpc-c benchmark, revision 5.11, 2010.

[49] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proc. of 1986 Int'l Conf. on Parallel Processing*, 1986.

[50] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[51] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[52] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. 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.

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

[54] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[55] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM SIGPLAN Notices*, volume 29, pages 230–241. ACM, 1994.

[56] M. DeVuyst, D. M. Tullsen, and S. W. Kim. Runtime parallelization of legacy code on a transactional memory system. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 127–136. ACM, 2011.

[57] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of lazy subscription. In *WTTM*, 2014.

[58] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. In *Transact 2008 workshop*, 2008.

[59] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[60] Dice, D. and Shavit, N. What Really Makes Transactions Faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006.

[61] N. Diegues and P. Romano. Self-tuning Intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing*, ICAC '14. USENIX Association, 2014.

[62] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying Transactional Memory. *Formal Aspects of Computing*, 25(5):769–799, 2013.

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

[64] A. Dragojevic and T. Harris. STM in the small: trading generality for performance in software transactional memory. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 1–14, 2012.

[65] Z. Du, C. Lim, X. Li, C. Yang, Q. Zhao, and T. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. *ACM SIGPLAN Notices*, 39(6):71–81, 2004.

[66] T. J. Edler von Koch and B. Franke. Limits of region-based dynamic binary parallelization. In *ACM SIGPLAN Notices*, volume 48, pages 13–22. ACM, 2013.

[67] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[68] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen. The polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, 1994.

[69] P. Felber, C. Fetzer, and T. 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.

[70] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 93–107, 2009.

[71] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *ACM SIGPLAN Notices*, volume 46, pages 458–469. ACM, 2011.

[72] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic scalable atomicity via semantic locking. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 31–41, 2015.

[73] M. Gonzalez-Mesa, E. Gutierrez, E. L. Zapata, and O. Plata. Effective transactional memory execution management for improved concurrency. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):24, 2014.

[74] T. Grosser, A. Groesslinger, and C. Lengauer. Polly: performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[75] R. Guerraoui and M. Kapalka. Opacity: A Correctness Condition for Transactional Memory. Technical report, EPFL, 2007.

[76] R. Guerraoui and M. 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.

[77] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2011.

[78] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, 2000.

[79] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, and E. Bu. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[80] L. Hammond, M. Willey, and K. Olukotun. *Data speculation support for a chip multiprocessor*, volume 32. ACM, 1998.

[81] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *in Proc. of ISCA*, page 102, 2004.

[82] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, (38), 2003.

[83] T. Harris, J. Larus, and R. Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[84] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[85] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 387–388, 2014.

[86] D. Heath, R. Jarrow, and A. Morton. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica: Journal of the Econometric Society*, pages 77–105, 1992.

[87] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364, 2010.

[88] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[89] M. Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 1–2, New York, NY, USA, 2006. ACM.

[90] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[91] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. volume 41, pages 253–262, New York, NY, USA, October 2006. ACM.

[92] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[93] M. Herlihy and J. E. 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.

[94] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In J. Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos,*

Cyprus, July 7-11, 2008, Proceedings, volume 5142 of *Lecture Notes in Computer Science*, pages 76–103. Springer, 2008.

[95] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60:151–157, November 1996.

[96] W. M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, 1993. 10.1007/BF01205185.

[97] R. Intel. Architecture instruction set extensions programming reference. *Intel Corporation, Feb*, 2012.

[98] T. Johnson. Characterizing the performance of algorithms for lock-free objects. *Computers, IEEE Transactions on*, 44(10):1194 –1207, Oct. 1995.

[99] D. Kanter. Analysis of haswells transactional memory. *Real World Technologies (Febuary 15, 2012)*, 2012.

[100] N. Karjanto, B. Yermukanova, and L. Zhexembay. Black-scholes equation. *arXiv preprint arXiv:1504.03074*, 2015.

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

[102] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *Computers, IEEE Transactions on*, 48(9):866–880, 1999.

[103] V. P. Krothapalli and P. Sadayappan. An approach to synchronization for parallel computing. In *Proceedings of the 2nd international conference on Supercomputing*, pages 573–581. ACM, 1988.

[104] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. 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.

[105] M. Lam and M. Rinard. Coarse-grain parallel programming in jade. In *ACM SIGPLAN Notices*, volume 26, pages 94–105. ACM, 1991.

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

[107] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[108] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[109] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT*, 2007.

[110] Z. Li, A. Jannesari, and F. Wolf. Discovery of potential parallelism in sequential programs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 1004–1013. IEEE, 2013.

[111] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167. ACM, 2006.

[112] V. Luchangco, M. Wong, H. Boehm, J. Gottschlich, J. Maurer, P. McKenney, M. Michael, M. Moir, T. Riegel, M. Scott, et al. Transactional memory support for c+. 2014.

[113] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Appl. Math.*, 25:145–153, September 1989.

[114] J. Mankin, D. Kaeli, and J. Ardini. Software transactional memory for multicore embedded systems. *SIGPLAN Not.*, 44(7):90–98, 2009.

[115] P. J. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *DSN*, pages 454–465, 2011.

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

[117] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA*, pages 67–74, 2008.

[118] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Reengineering for parallelism: An entry point into plpp (pattern language for parallel programming) for legacy applications. In *Proceedings of the Twelfth Pattern Languages of Programs Workshop (PLoP 2005)*, 2005.

[119] A. Matveev and N. 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*, pages 59–71. ACM, 2015.

[120] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[121] V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for java stm. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 314–325, 2008.

[122] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, pages 171–179, 2003.

[123] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

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

[125] M. Mohamedin, B. Ravindran, and R. Palmieri. Bytestm: Virtual machine-level java software transactional memory. In *Coordination Models and Languages*, pages 166–180. Springer, 2013.

[126] M. A. M. Mohamedin. On optimizing transactional memory: Transaction splitting, scheduling, fine-grained fallback, and numa optimization. 2015.

[127] M. Moir. Practical implementations of non-blocking synchronization primitives. In *In Proc. of 16th PODC*, pages 219–228, 1997.

[128] K. E. Moore. Thread-level transactional memory. In *Wisconsin Industrial Affiliates Meeting*. Oct 2004. Wisconsin Industrial Affiliates Meeting.

[129] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *In Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[130] J. E. B. Moss. Open nested transactions: Semantics and support. In *In Workshop on Memory Performance Issues,*, 2005.

[131] J. E. B. Moss, N. D. Griffeth, and M. H. Graham. Abstraction in recovery management. In *ACM SIGMOD Record*, volume 15, pages 72–83. ACM, 1986.

[132] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63:186–201, December 2006.

[133] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[134] S. C. Müller, G. Alonso, A. Amara, and A. Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 645–659, Broomfield, CO, Oct. 2014. USENIX Association.

[135] A. MySQL. *MySQL: the world's most popular open source database*. MySQL AB, 1995.

[136] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 68–78, 2007.

[137] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of the 45th annual Design Automation Conference*, pages 574–579. ACM, 2008.

[138] C. Nvidia. Compute unified device architecture programming guide. 2007.

[139] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[140] C. D. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *Computers, IEEE Transactions on*, 37(8):991–1004, 1988.

[141] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 1–12, New York, NY, USA, 2003. ACM.

[142] C. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *ACM Sigplan Notices*, volume 40, pages 269–279. ACM, 2005.

[143] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an stm. In *ACM Sigplan Notices*, volume 44, pages 163–172. ACM, 2009.

[144] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 65–76, New York, NY, USA, 2010. ACM.

[145] R. Ramaseshan and F. Mueller. Toward thread-level speculation for coarse-grained parallelism of regular access patterns. In *Workshop on Programmability Issues for Multi-Core Computers*, page 12, 2008.

[146] L. Rauchwerger and D. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *SIGPLAN Not.*, 30:218–232, June 1995.

[147] K. Ravichandran, A. Gavrilovska, and S. Pande. Destm: Harnessing determinism in stms for application development. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 213–224, 2014.

[148] Y. Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *VLDB*, volume 92, pages 292–312, 1992.

[149] J. Reinders. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 2013.

[150] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In S. Dolev, editor, *Distributed Computing*, Lecture Notes in Computer Science, pages 284–298. Springer Berlin / Heidelberg, 2006.

[151] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber. From causal to z-linearizable transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 340–341, New York, NY, USA, 2007. ACM.

[152] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. 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.

[153] C. J. Rossbach, Y. Yu, J. Currey, J. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 49–68. ACM, 2013.

[154] E. Rotenberg and J. Smith. Control independence in trace processors. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 4–15. IEEE Computer Society, 1999.

[155] W. Ruan, Y. Liu, and M. Spear. Stamp need not be considered harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*, 2014.

[156] W. Ruan, Y. Liu, and M. F. Spear. Transactional read-modify-write without aborts. *TACO*, 11(4):63:1–63:24, 2014.

[157] W. Ruan and M. Spear. An opaque hybrid transactional memory. 2015.

[158] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *ACM SIGPLAN Notices*, volume 34, pages 72–83. ACM, 1999.

[159] G. Rünger and M. Schwind. Parallelization strategies for mixed regular-irregular applications on multicore-systems. In *Advanced Parallel Processing Technologies*, pages 375–388. Springer, 2009.

[160] M. M. Saad, M. Mohamedin, and B. Ravindran. Hydravm: Extracting parallelism from legacy sequential code using STM. In *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*, 2012.

[161] M. M. Saad, R. Palmieri, A. Hassan, and B. Ravindran. Extending tm primitives using low level semantics. In *SPAA '16: The 28th ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2016. ACM.

[162] M. M. Saad, R. Palmieri, A. Hassan, and B. Ravindran. On extending tm primitives using low level semantics. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2016.

[163] M. M. Saad, R. Palmieri, and B. Ravindran. Lerna: Transparent and effective speculative loop parallelization. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2016.

[164] M. M. Saad, R. Palmieri, and B. Ravindran. On ordering transaction commit. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 46:1–46:2, 2016.

[165] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06*, pages 187–197, Mar 2006.

[166] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[167] J. H. Saltz, R. Mirchandaney, and K. Crowley. The preprocessed doacross loop. In *ICPP (2)*, pages 174–179, 1991.

[168] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC '04: Proceedings of Workshop on Concurrency and Synchronization in Java Programs.*, NL, Canada, 2004. ACM.

[169] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[170] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[171] M. Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[172] G. S. Sohi, S. E. Breach, and T. Vijaykumar. Multiscalar processors. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 414–425. ACM, 1995.

[173] M. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. Scott, C. Ding, and P. Wu. Fastpath speculative parallelization. *Languages and Compilers for Parallel Computing*, pages 338–352, 2010.

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

[175] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[176] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 2–13. IEEE, 1998.

[177] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. *A scalable approach to thread-level speculation*, volume 28. ACM, 2000.

[178] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: runtime adaptive parallel execution. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*, page 7. ACM, 2013.

[179] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 356–369. IEEE, 2007.

[180] Transactional Memory Specification Drafting Group. Draft specification of transactional language constructs for C++, version 1.1, February 2012. Available `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3725.pdf`.

[181] J. Tsai and P. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on*, pages 35–46. IEEE, 1996.

[182] N. A. Vachharajani. *Intelligent speculation for pipelined multithreading*. PhD thesis, Princeton, NJ, USA, 2008. AAI3338698.

[183] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 389–400. ACM, 2010.

[184] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 185–196. ACM, 2008.

[185] C. Von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP*, pages 79–89, 2007.

[186] P. Wu, A. Kejariwal, and C. Caşcaval. Compiler-driven dependence profiling to guide program parallelization. *Languages and Compilers for Parallel Computing*, pages 232–248, 2008.

[187] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[188] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 76–86, 2015.

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

[190] L. Zhang, V. K. Grover, M. M. Magruder, D. Detlefs, J. J. Duffy, and G. Graefe. Software transaction commit order and conflict management, May 4 2010. US Patent 7,711,678.

[191] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[192] H. P. Zima, H.-J. Bast, and M. Gerndt. Superb: A tool for semi-automatic mimd/simd parallelization. *Parallel computing*, 6(1):1–18, 1988.