Concurrency in Transactional Memory

Vincent Gramoli

Concurrency in Transactional Memory

Thèse d'habilitation à diriger des recherches presented to

UPMC - Sorbonne University

in the subject of

Computer Science

by

Vincent Gramoli

on

July 6th, 2015

Jury

Albert COHEN	Senior Researcher, INRIA	Referee
Tim HARRIS	Senior Researcher, Oracle Labs	Referee
Maged MICHAEL	Senior Researcher, IBM Watson Research Center	Referee
Anne-Marie KERMARREC	Senior Researcher, INRIA	Examiner
Gilles MULLER	Senior Researcher, INRIA	Examiner
Pierre SENS	Professor, UPMC	Examiner

Contents

Abstract

This document is presented in fulfilment of the degree of *Habilitation à Diriger des Recherches* and aims at presenting, in a coherent form, my main scientific results, collaborations and supervision of research during the last 8 years in the domain of transactional memory where I have sought to contribute to the advancement of research, and the future perspectives that my contribution leads to. The document comprises three main chapters that are summarized as follows.

For the last decade, manufacturers have increased the number of processors, or *cores* in most computational devices rather than their frequency. To increase the performance of applications, one must write *concurrent programs*, programs divided into subroutines that cores can execute simultaneously, a task considered very difficult. My contribution has been to facilitate the adoption of transaction, a synchronization abstraction to simplify concurrent programming. First, I contributed to the integration of transactional memory (TM) in two mainstream programming languages and demonstrated empirically that software TM could leverage multicore and manycore platforms. Second, I defined *concurrency* as a metric capturing the variety of sequences of shared memory accesses a concurrent program exports and exploited this metric to identify a concurrency limitation that is inherent to the transaction abstraction. Third, I proposed a solution to overcome this concurrency limitation, developed a *polymorphic transactional memory* offering multiple transaction models for both expert and novice programmers, and proposed a redesign of a transactional data structure to improve the performance of applications using software transactional memory.

My research results open up three research directions related to the systematic comparison of synchronization techniques, the generalization of the structure redesign principles to a larger class of concurrent data structures and the definition of contention that has become a predominant cause of performance drop.

Acknowledgments

I am grateful to the members of my HDR jury: Albert Cohen, Tim Harris and Maged Michael for having accepted to review my HDR thesis and having actively participated to my defence. I wish to thank the examiners, Anne-Marie Kermarrec and Pierre Sens who examined my work and Gilles Muller for chairing the jury.

I am grateful to my collaborators, colleagues and students, who contributed to the results presented in this thesis. First, I wish to thank Dr. Crain, Dragojevic, Harmanci, Letia, Ravi who were all PhD students at the time of our collaboration and who all contributed to advancing the research presented here along with completing successfully their PhD degree. Second, my co-authors and friends for all the fruitful discussions we had on the topic of transactional memory including Pascal Felber, Rachid Guerraoui, Hagit Attiya, Petr Kuznetsov and Alessia Milani.

I would like to thank my wife, children and family for supporting my research over these 11 last years.

1. Introduction

Modern processors are not faster than the processors we used fifteen years ago.[1] For the last decade, manufacturers have increased the number of processors, or *cores*, in most computational devices rather than their frequency. This trend led to the advent of *chip multiprocessors* that offer nowadays between tens [146] to a thousand cores on the same chip [121].

Concurrent programming, which is the art of dividing a program into subroutines that cores execute simultaneously, is the only way for developers to increase the performance of their software. These multicore machines adopt a concurrent execution model where, typically, multiple threads *synchronize* with each other to exploit cores while accessing in-memory shared data. To continue the pace of increasing software efficiency, performance has to scale with the amount of concurrent threads accessing shared data structures. The key is for new synchronization paradigms to not only leverage concurrent resources to achieve scalable performance but also to simplify concurrent programming so that most programmers can develop efficient software.

As it is often the case when facing such performance challenges, researchers attempted to weaken the targeted consistency to improve performance, hence defining relaxed consistency criteria [132] and considering that data structures remain correct despite violating some invariants. For example, the queue of the Intel® Threading Building Blocks library is considered correct even though it violates the first-in-first-out (FIFO) specification of a queue. Similarly, a recent relaxed stack was designed to achieve scalability on an Intel machine comprising 80 cores [45]. Finally, the `java.util.ConcurrentLinkedQueue.size()` is documented as being "not so useful" in concurrent programs as it may return an incorrect result.

The drawback of consistency relaxation is to make concurrent programming a difficult task only reserved to experts. Liskov reminded recently the importance of encapsulation in programming languages [93], this lets Bob *reuse* straightforwardly an *Abstract Data Type* (ADT) designed by Alice. Encapsulation has been extensively used to make "sequential" object oriented programs reusable [114]. The reason of this reusability, is that the ADT exposes a simple semantics that Bob can reason about when designing his application, even without understanding how the type was implemented. In a concurrent context, this simple semantics is provided by *Concurrent Data Types* (CDTs) and can be described as a sequential specification that is easy to reason about. As soon as a concurrent data structure stops guaranteeing the sequential specification of the type it was expected to implement, it becomes difficult for Bob to understand if the slightly different implemented type is still well-suited for his application.

Transactional memory has been instrumental in simplifying concurrent programming. Protecting the operations offered by a sequential implementation of a type with classic transactions guarantees that the resulting concurrent implementation will be atomic (i.e., linearizable [91]), hence ensuring that any execution of the concurrent implementation is equivalent to one that satisfies the sequential specification of the original type. In other words, if Alice delimits with transactions the `pop` and `push` operations of a sequential implementation of a queue type, then the resulting concurrent implementation is guaranteed to implement a queue type as well—its FIFO property being preserved. Bob does not have to be an expert in concurrent programming to know whether Alice's queue can be used in his application. In particular, Bob does not have to look at Alice's code if he uses transactional memory to develop his concurrent application based on Alice's library.

Transactional memory is, however, not always efficient. First hardware transactional memories (HTMs) are limited by the hardware and require a software fall-back mechanism to guarantee that transactions can execute in software when they cannot be fully executed in hardware. Second, software transactional memories (STMs) raised some doubts about their capability to leverage multicore machines [27], not to mention manycore machines. Part of our research was to clarify these folklore

[1] The iPad Air 2 processor runs as fast (1.5 GHz) as the Willamette-based Pentium 4 processor from November 2000.

believes with more recent experiments [46, 64]. We observed, however, that the transaction concept at the heart of any transactional memory suffers from an inherent expressiveness limitation. We showed that this limitation translated into concurrency limitations in simple data structure workloads [60, 61], hence outlining a real performance concern that could affect the adoption of transactions.

To address this issue, we proposed a novel synchronization paradigm that slightly changes the classic notion of transactional memory to offer multiple transactions to the programmers [54, 59–61]. The key idea is to relax the semantics of the transaction without affecting the consistency of the application by proposing multiple transaction *forms* with different semantics. It allows an expert programmer, say Alice, to exploit the inherent concurrency of her application by choosing the appropriate transaction semantics in her application. By preserving the sequential specification of the type our, so-called *polymorphic*, solution guarantees that Bob can straightforwardly reuse Alice's concurrent data type. Moreover, our polymorphic solution allows a novice programmer like Bob to use the default transaction semantics, hence retaining the simplicity of transactions.

To improve performance a step further, we worked at relaxing the data structure invariants in addition to offering relaxed transactional models [33–35, 59]. Again by never relaxing the consistency of the implemented type we preserve the simplicity of concurrent programming. As opposed to type invariants, data structure invariants simply impose requirements on the layout of the data. By relaxing data structure invariants, we violate the step complexity of data structures during transient periods of high contention to boost the performance. In fact, by not enforcing threads to restructure the data layout upon concurrent updates, we are able to restrict the contention to localized subparts of the structure, hence offering a solution to design efficient transactional data structures.

1.1 Contributions

This document presents my main research contributions over the last 8 years. The work presented here is the result of collaborations with colleagues from Université de Neuchâtel and EPFL in Switzerland, Université de Rennes 1 and INRIA in France, and NICTA and University of Sydney in Australia. In particular, this includes results from the work of 5 PhD students that I helped supervising and who all successfully graduated. My contributions are as follows:

1. My first contribution was to help making the notion of transactional memory mainstream [46,56,59, 64,75,76]. First, I contributed with multiple partners to the design of the first transactional memory stack by developing benchmarks to test programming language supports [75] and evaluate their performance [59], by developing transaction support with a Java compiler [76] and by testing the support of transactional memory in the TM branch of GNU gcc [56]. Second, I tested extensively the performance of STM with manual and compiler instrumentations, and explicit and transparent privatizations on multicores [46] and with message passing on top of non-cache-coherent machines, also called *manycores* [64].

2. My second contribution is the identification of concurrency limitations in transactional memory [60, 65, 66, 77]. First, I defined the concept of input schedule that was instrumental in designing a concurrency metric to compare (i) the concurrency one could obtain from different transactional algorithms [65] and (ii) the concurrency one could obtain from implementations synchronized with transactions and locks [66]. Second, I identified differences among existing transactional memory algorithms by using a unit-testing framework [77] but showed that, regardless of the semantics of these algorithms, there exist workloads where transactions cannot offer a level of concurrency as high as locks [60].

3. My third contribution is a solution to improve the performance of transactional applications [33, 54, 61, 63]. First, I proposed a relaxed transactional model to bypass the concurrency limitation of

transactions in some search data structure applications [54]. Second, I investigated techniques to compose relaxed transactions with each other [63] and to combine multiple relaxed transactional semantics within the same transactional memory, so called polymorphic [61]. Third, I proposed to improve concurrency further by rethinking data structures to reduce the size of the transactions and relaxing the data structure invariants to minimize contention [33].

These contributions helped making transactional memory an off-the-shelf tool suitable for differently skilled programmers. In particular, transactional memory is now integrated in programming languages, it offers scalable performance in various applications both on multicores and manycores. Our polymorphic transactional memory improves the performance of classic transactional memories without annihilating simplicity and our redesign of an existing data structure in the context of (polymorphic) transactional memory opens up new research directions.

1.2 Transaction

The transaction abstraction comprises the mechanisms used to synchronize the accesses to data shared by concurrent threads or *processes*. The abstraction dates back to the 70's when it was proposed in the context of databases to ensure the *consistency* of shared data [50]. This consistency was determined with respect to a sequential behavior through the concept of *serializability* [123]: concurrent accesses need to behave as if they were executing sequentially. Since then, researchers have derived other variants (like isolation [134] and opacity [73]) applicable to different transactional contexts.

The transaction abstraction was considered for the first time as a programming language construct in the form of *guards* and *actions* in [105]. Then it was adapted to various programming models, e.g., Argus [104], Eden [9] and ACS [69]. The first hardware support for a transactional construct was proposed in [98]. It basically introduced parallelism in functional languages by providing synchronization for multiple memory words. Later, the notion of transactional memory was proposed in the form of hardware support for concurrent programming to remedy the difficulties of using locks, e.g., priority inversion, lock-convoying and deadlocks [89].

Software transactions were originally designed as a reusable and composable solution to execute a set of shared memory accesses fixed prior to the execution [133]. Later, they were applied to handle the case where the control flow was not predetermined [88]. Bob's transaction can invoke the transactions of Alice without risking to deadlock during an execution [80].

There are various programming issues with the use of STM (e.g., ensuring weak or strong atomicity, support for legacy binary code, etc). The alternative concurrency programming approaches like fine-grained locking or lock-free techniques are, however, not easier to use. Such comparisons have already been discussed in [54, 68, 78, 88, 122, 133] and our work does not focus on this aspect. For example, Pankratius and Adl-Tabatabai performed a case study on programming teams who developed parallel programs from scratch using either locks or transactions. They showed that it was easier to read the code based on transactions than the one based on locks [122].

1.3 Data Type Implementation

ADTs are reusable as they promote (a) *extensibility* when an ADT is specialized through, for example, inheritance by overriding or adding new methods, and (b) *composability* when two ADTs are combined into another ADT whose methods invoke the original ones. Key to this reusability is that there is no need to know the internals of an ADT to reuse it: its interface suffices. With the latest technology development of multi-core architectures many programs are expected to scale with a large number of cores: ADTs need thus to be shared by many threads, hence becoming what we call concurrent data types (CDTs).

We consider the *set* data type as a running example as it is a simple data type and can be easily generalized into a multi-set or a collection and a dictionary or a map data type. For the sake of simplicity, the states of the set data type are all finite subsets of $\mathbb{Z}$. For any $v \in \mathbb{Z}$, the set data type exports `insert`(v) that adds v to the set and returns `true` if v was not already present, otherwise it returns `false`; `remove`(v) that deletes v from the set and returns `true` if v was present, otherwise it returns `false`; and `contains`(v) that returns `true` if v is present or `false` otherwise. The sequential specification of the set is detailed later (§§§3.1.1).

As we are interested by concurrency, we chose a simple concurrency-friendly data structure [137] to implement the set data type, a sorted linked list. This sorted linked list that implements the set data type serves as a running example, referred to as a *list-based set* in the sequel. We design two sentinel nodes: the first element of the list *head*, storing value $-\infty$, and the last element of the list *tail*, storing value $+\infty$. Initially, when the set is empty, $head.next = tail$.

1.4 Chip Multiprocessors

To measure the performance of software transactional memory, we use common muticore machines with cache coherence as well as less conventional manycore machines without cache coherence. A *manycore* is a processor that embeds a large number of simpler cores than a multicore to maximize overall performance while minimizing energy consumption [20]. The backbone of a manycore is a network-on-chip, which interconnects all cores and carries the memory traffic. Every core has private caches, however, a manycore might have either a limited or no hardware cache coherence at all. Therefore, this on-die interconnection network provides the programmer with efficient message passing. In order to increase the memory bandwidth, a manycore processor is connected to multiple memory controllers [2]. These controllers provide both the private and the (non-coherent) shared memory of the cores.

All experiments that led to the results presented in the remainder were run on the machines listed in Table 1. The Intel Xeon machine uses the QuickPath Interconnect for the MESIF cache coherence protocol to access the distant socket L3 cache faster than the memory. The latency of access to the L1D cache is 4 cycles, to the L2 cache is 10 cycles and to the L3 cache is between 45 and 300 cycles depending on whether it is a local or a remote access [95]. The latency of accesses to the DRAM is from 120 to 400 cycles. The Sun Microsystems UltraSPARC T2 features 64 hardware threads running on 8 cores, thanks to simultaneous multi-threading, at 1.165 GHz. This machine has 32 GB of memory and runs Solaris 10.

We used three different AMD machines all with AMD Opterons. For inter-socket communication, these machines rely on AMD's HyperTransport and feature the HyperTransport Assist [32] probe filter that acts as a directory in order to reduce the cache-coherence traffic. The first AMD machine comprises 4 sockets of 16 cores with 128GB of memory, the second one has 8 sockets of 6 cores also with 128 GB while the third one has 4 sockets of 4 cores with 4GB of memory. The AMD machines use HyperTransfer with a MOESI cache coherence protocol for inter-socket communication, they have a slightly lower latency to access L1 and L2 caches and a slightly higher latency to access the memory [29] than the Intel Xeon machine.

The Intel Single-chip Cloud Computer (SCC) [92] is an experimental manycore platform that embeds 48 non-cache-coherent cores on a single die. Its architecture represents a 6×4 bidimensional mesh of tiles, each tile comprising two x86 cores. Every core has 32 KB of L1 cache, a separate 256 KB L2 cache, and provides one globally accessible atomic test-and-set register. In addition, each tile has 16 KB of SRAM, called the message passing buffer (MPB), for implementing message passing. Finally, the SCC processor includes 4 memory controllers, with a total of 32 GB of memory. Every core uses a partition of this memory as its local RAM and the remaining can be allocated as shared memory. An important characteristic of the SCC is the lack of hardware cache coherence. The coherence of the shared memory and the MPB must be handled in software. We use the SCC with the default performance settings: 533/800/800 MHz for core/mesh/memory frequencies, respectively.

Man.	Processor	Archi.	#HW th.	#cores	#sockets	Memory	GHz	Reference	$ Coherence
AMD	Opteron	CISC	64	64	4	128 GB	1.4	[59]	yes
AMD	Opteron	CISC	48	48	8	128 GB	2.1	[64]	yes
AMD	Opteron	CISC	16	16	4	8 GB	2.2	[46]	yes
Intel	Xeon	CISC	32	16	2	132 GB	2.1	[43,59]	yes
Intel	SCC	CISC	48	48	1	32 GB	0.53	[64]	no
Sun	UltraSPARC T2	RISC	64	8	1	32 GB	1.165	[59,61]	yes
Tilera	TileGx	RISC	36	36	1	32 GB	1.2	[64]	tunable

Table 1: The multicore/manycore configurations used in our experiments, with the manufacturer, the processor, the type of architecture, the number of hardware threads, cores and sockets, the memory, the clock frequency, the publications where the experiments appeared and whether they support cache coherence

The Tilera TILE-Gx36™ [138] is a manycore platform that offers both cache coherence and message passing in hardware. The TILE-Gx36 features 36 identical processor cores (tiles) interconnected with Tilera's on-chip network. The TILE-Gx is a chip multiprocessor whose architecture is similar to the one of Figure 2(b). Each tile runs at 1.2 GHz and is a full-featured processor, including integrated L1 (64 KB) and L2 (256 KB) caches. The TILE-Gx family incorporates Tilera's DDC™ technology (dynamic distributed cache) that implements efficient cache coherence. In short, the L2 caches of all tiles are globally accessible, thus forming a distributed LLC that is kept consistent using a directory that is partitioned across all tiles.

The environment can be thus modeled as a fully distributed system whose *nodes*, which represent cores, are fully connected and can communicate with each other using asynchronous messages. We assume that the communication links between nodes are reliable. In addition, we assume that nodes are uniquely identified and non-faulty in that they respect their code specification and do not crash. Note also that this model is sufficiently general to capture both homogeneous and heterogeneous manycores [19].

1.5 Roadmap

We first introduce transactional memory as a programming paradigm that evolved from a research toy to a mature technology ported to multicore and manycore computers and integrated into compilers to help programmers design software that exploit the growing computational resources of modern machines (§2). We however reason formally about the concurrency of transactional software and identify limitations that are inherent to the classic transactional model (§3). To address this concurrency limitation while guaranteeing consistency, we propose alternative transactional models and a new method to design transaction-based data structures (§4). Finally, we envision comparing other synchronisation techniques by extending our concurrency model to other synchronisation techniques, by generalizing our method of designing transaction-based data structures to the method of designing concurrent data structures and by measuring their performance (§5).

2. Transactional Memory: A Mature Technology

Transactional memory (TM) was proposed in the form of hardware support for concurrent programming to remedy the difficulties of using locks, e.g., priority inversion, lock-convoying and deadlocks [89]. Since the advent of chip multiprocessors, the very notion of transactional memory has become an active topic of research[2]. Hardware implementations of transactional systems [89] turned out to be limited by specific constraints the programmer could only "abstract away" from using unbounded hardware transactions. Purely hardware implementations are however complex solutions that most industrials are no longer exploring. Instead, a hybrid approach was adopted by implementing a best-effort hardware component that needs to be complemented by *software transactions* [39].

2.1 Integration in Programming Languages [56]

Programming with transactions shifts the inherent complexity of concurrent programming to the implementation of the transaction semantics that must be achieved once and for all. Thanks to transactions, writing a concurrent application follows a divide-and-conquer strategy where experts have the complex task of writing a live and safe transactional system with an unsophisticated interface so that the novice programmer simply has to write a transaction-based application, namely, delimit regions of sequential code, typically using a transaction{...} block.

Transactional memory executes in hardware, software, or as a combination of the two. Because transaction block constructs are typically provided at the programming language level, transactional memory support spans the complete computer stack, from applications to hardware, encompassing language extensions, compilers, libraries, transactional memory runtime, and operating system. The VELOX project[3], a 3-year European research project starting in 2008 and entitled *An Integrated Approach to Transactional Memory on Multi and Many-core Computers*, led to the development of a transactional memory stack spanning hardware, compiler and application levels. As a scientific manager of this project, my roles at EPFL and the University of Neuchâtel were to develop benchmarks to test the integration of transactions at all the levels of the stack, to design a new transactional model and to integrate the transaction language constructs in a Java compiler.

As Figure 1 illustrates, the transactional memory stack resulting from the VELOX project consists of several basic components. To better exploit transactional memory, system libraries must be adapted to execute speculatively inside transactions despite performing potentially unsafe operations (for example, I/O). Where applicable, a developer can also replace locks with transactions within libraries for better performance. The transactional memory runtime is the transactional memory integrated stack's central component. It implements the transactional memory's synchronization logic. Operating system extensions can help improve transactional memory's performance for some tasks relating to the system as a whole (scheduling, for example), in particular because transactional workloads coexist with traditional workloads.

Language extensions and APIs are the most visible aspects of transactional memory for programmers. Although programmers could add transactional memory support to applications using explicit library calls or declarative mechanisms (such as annotations), this approach is not satisfactory for large systems. This support relies on coding conventions, can lead to intricate code and is often error prone. At the other end of the spectrum, automated source or binary code instrumentation work on simple examples, but are difficult to extend to realistic code. Adding new language constructs with well-defined semantics is the soundest approach for importing transactional memory support into existing

[2] http://www.cs.wisc.edu/trans-memory/biblio/list.html.

[3] http://www.velox-project.eu.

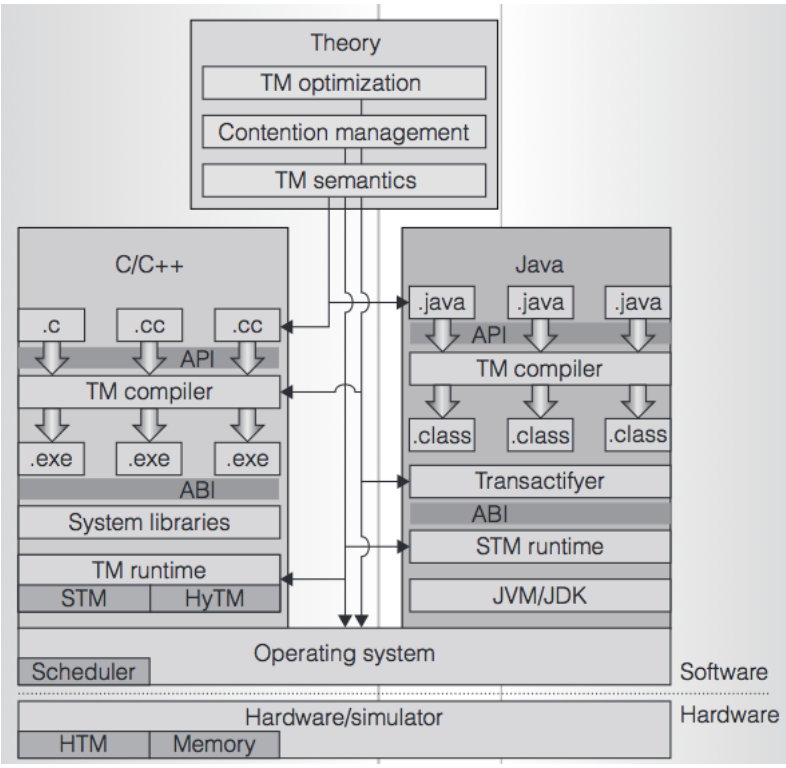


Figure 1: The transactional memory stack supports two families of programming languages: C/C++ and Java and the associated tool chains and libraries; the VELOX project has studied a set of crosscutting research challenges to drive the design and development of the stack components

languages. Compiler support is necessary for implementing an efficient transactional memory stack in a way that is transparent to the programmer. For instance, it lets us identify transactional load and store operations on shared data and map them to the underlying transactional memory without requiring the programmer to explicitly mark these operations. Typically, this transparency relieves the programmer from the burden of instrumenting manually each read and write access within all the nested functions of her transaction, which has two main disadvantages: (i) it makes the program error-prone due to manual instrumentation, (ii) it limits optimizations as the compiler has no insight on the transaction access pattern. In the VELOX project, we have developed programming language extensions, by concentrating primarily on C/C++ and Java.

2.1.1 Transaction support in C/C++

The first attempts to support TM in production compilers led to the definition of dedicated transactional language constructs in a C++ specification draft [141], resulting originally from the collaboration of Intel, IBM and Sun Microsystems, acquired later by Oracle. This draft describes the constructs to delimit a compound statement that is identified by the compiler as a transaction, in addition to resulting subtleties. For several years, this specification had been reviewed and discussed by academics and industrials on the tm-language mailing list[4], for the sake of language expressiveness and compliance with existing TM

[4] http://groups.google.com/group/tm-languages.

systems and other languages.

On the C/C++ side, my task in the VELOX project was to help Red Hat identifying bugs in the TM branch of gcc[5] before it could be released mainstream as part of v4.7. At that time, there existed other compilers with TM support like the prototype DTMC[6] [55] and the Intel® C++ STM compiler [120] (icc)[7]. In C, a transaction is simply delimited using the block

```
__tm_atomic{ ... }
```

while in C++ a transaction is delimited by a `__transaction{ ... }` block where `__transaction{` (or equivalently `__transaction[[atomic]]{`) indicates the point in the code where the corresponding transaction should start. The closing bracket `}` indicates the point in the code where the corresponding transaction `commit` should be called. Within this block, memory accesses are instrumented by the compiler to call the transactional `read` and `write` wrappers. More precisely, the binary files, produced by the compiler, call a dedicated TM runtime library through an appropriate application binary interface (ABI) specified in [96]. This ABI is used for both C and C++ and has been optimized for the Linux OS and x86 architectures to reduce the overhead of the TM calls and to allow fast accesses to thread-specific metadata shared by existing TMs.

Transaction nesting, which consists in enclosing a transaction block into another, is allowed and the `[[outer]]` keyword is explicitly used to indicate that a transaction cannot be nested inside another. Generally, it is not allowed to redirect the control flow to some point in the context of a transaction, but exceptions can be raised within the context of transaction to redirect the control-flow outside the transaction context by propagating the exception.

Irrevocable transactions do not execute speculatively and are used to execute actions that cannot be rolled back once executed, this is typically necessary in cases where an action has some external side-effects like I/O have. Attributes in C++1x-style indicate whether the transaction executes speculatively as by default `__transaction[[atomic]]{}` or has to execute without being aborted `__transaction[[relaxed]]{}`, say in *irrevocable* mode. Only irrevocable transactions can execute calls with irrevocable side-effects, and for example

```
[[transaction_unsafe]] void fire_missile{};
```

declares a `fire_missile` function that can only be called in an irrevocable transaction. The attribute `[[transaction_safe]] void do_work{};` is especially used to indicate the opposite, that function `do_work` does not have to be called in an irrevocable transaction and can execute speculatively as part of a transaction prone to abort.

2.1.2 Transaction support in Java

On the Java side, we supported transactions by adding an atomic block construct to declare transactions and a few extra keywords to handle operations such as explicit abort and retry. We have developed and released an open source transactional compiler, TMJava[8], that was used successfully for exception handling with transactional memory [76]. TMJava processes Java source code with transactional constructs `atomic{...}` and generates pure Java classes that will subsequently be instrumented by the Deuce framework [99] to produce a transactional application whose conflicts are detected at the granularity of the field, rather than the object itself.

Initially, annotations and bytecode instrumentation helped supporting TM in Java to instrument transactional accesses either at load-time or statically, prior to execution. Multiverse[9] and Deuce [99] are

[5] http://www.velox-project.eu/software/gcc-tm.
[6] http://www.velox-project.eu/software/dtmc.
[7] http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/.
[8] http://www.tmware.org/tmjava.
[9] http://multiverse.codehaus.org.

two such JVM agents that instrument transactional accesses of the annotated bytecode resulting from a concurrent program.

Multiverse distinguishes `@TransactionalObject` and `@TransactionalMethod` annotations that apply respectively to Java objects and methods. All instance methods of an annotated object are thus instrumented transactions and annotated methods allow to specify which methods of a non-annotated object are transactions. Additionally, annotating a method of an already annotated objects allows to differentiate explicitly read-only methods from update methods using `@Transactional(readonly = true)`. Such differentiation is useful for the underlying TM to optimize the validation of a read-only transaction that commits.

Deuce instruments methods annotated with the `@Atomic` keyword and uses a clear interface a TM should provide: `begin` (namely `init`), `read` (namely `onReadAccess`), `write` (namely `onWriteAccess`), `commit` methods and additional `beforeReadAccess` and `abort` (namely `rollback`) methods. The current distribution features classical transactions of state-of-the-art software TMs developed collaboratively like TL2 [40], LSA [125], NOrec [36] but also $\mathcal{E}$-STM [54] that combines classical and elastic transactions.

The aforementioned bytecode instrumentation frameworks could not consider an arbitrary compound statement as a transaction because they were constrained by the annotation mechanism. As a consequence, obtaining a close interaction between transactional memory and a compiler-generated code such as exception handling was difficult, if not impossible.

To fully support TM in Java, we developed a precompiler, TMJava, that remedies this limitation by extending Java with transactional blocks. More specifically, TMJava supports the `__transaction{...}` language construct in Java and outputs a purely Java annotated program whose bytecode gets instrumented using a backend instrumentation framework like Deuce. TMJava has been instrumental to apply TMs to coordinated exception handling in Java [76]. In this case, the failure atomicity guaranteed by transactions is useful for recovering from an inconsistent state even in a concurrent environment.

Since March 2012, TM has been supported in probably the mostly deployed compiler collection, gcc, and TMJava allows to add support for transactions in Java. Other languages support transactions as well. For example, the Glasgow Haskell Compiler provides support for transactions in Haskell [81] while Multiverse offers support for TM in Java and has been used in Scala [24]. While TM has been integrated in a complete stack spanning software, operating systems and hardware, it generally requires a software component to avoid the limitation of hardware and this software component induces significant overheads. It is thus crucial to measure the inherent cost of software transactional memory to understand whether it can truly leverage chip multiprocessors.

2.2 Leveraging Multicores [46]

A study on the performance of software transactions questioned their ability to leverage multicore architectures [27, 28]. In particular, this study showed empirically that applications synchronized with STM perform worse than the corresponding sequential applications without synchronization and running sequentially, and concluded that STM is only a "research toy". We revisited these results in the light of a systematic research effort we conducted to understand the causes of performance variation in concurrent executions. Our conclusions were surprisingly different from the observations of the initial study as we concluded that the same applications synchronized with STM actually outperforms the corresponding sequential applications in various settings on existing chip multiprocessors [46].

Many research papers have already reported that STM performance scale with the increasing number of threads on various benchmarks [4, 5, 10, 26, 40–42, 47, 53, 78, 87, 103, 107, 108, 120, 125–127, 134, 136]. Most of this work, however, does not compare STM to sequential code, thus ignoring the fundamental question of whether STM can be a viable option for actually speeding up the execution of applications on multicore machines. The STAMP macro-benchmark suite was used to test STM performance against

sequential performance [26]. STAMP consists of 8 different applications, summarized at the top of Table 2, that can be configured with a series of parameters to define real-world workloads. The evaluation of STAMP showed that STM outperforms sequential code, but only using a hardware simulator. We recently explored the speedup of synchronisation techniques over sequential performance, but without evaluating compiler instrumentation or privatization techniques [59].

The previous study [28] was the first to focus on the comparison of STM performance against sequential performance on real hardware. To this end, they used a micro-benchmark as well as three of the STAMP transactional applications, (i) `kmeans` for partition-based clustering, (ii) `vacation` for booking flights, rooms and cars, and (iii) `genome` for gene sequencing. All these experiments were performed using up to 8 threads on a quad-core hyper-threaded CPU. After getting the details of experimental settings from the authors, we noticed that their experiments did not use the default STAMP parameters. By contrast, the configurations they used significantly increase the contention in the three tested STAMP applications, by reducing the size of the shared data in `kmeans` and `vacation` and reducing the fraction of read-only transactions in `genome`.

Overheads of software transactional memory. Software transactional memory is not a panecea. In particular, there exist three forms of overheads:

1. **Synchronization costs.** Each read (or write) of a memory location from inside a transaction is performed by a call to an STM routine for reading (or writing) data. With sequential code, these accesses are performed by a single CPU instruction. STM read and write routines are significantly more expensive than corresponding CPU instructions as they, typically, have to maintain book-keeping data about every access. Most STMs check for conflicts with other concurrent transactions, log the access, and in case of a write, log the current (or old) value of the data, in addition to reading or writing the accessed memory location. Some of these operations use expensive synchronization instructions and access shared metadata, which further increases their costs. All of this reduces single-threaded performance when compared to sequential code.

2. **Compiler over-instrumentation.** To use an STM, programmers need to insert STM calls for starting and ending transactions in their code and replace all memory accesses from inside transactions by STM calls for reading and writing memory locations. This process, called *instrumentation*, can be *manual*, when the programmers manually replace all memory references with STM calls, or can be performed by an STM *compiler*. With a compiler, programmers only need to specify which sequences of statements have to be executed atomically, by enclosing them in transactional blocks. The compiler generates code that invokes appropriate STM read/write calls. While using an STM compiler significantly reduces programming complexity, it can degrade performance of resulting programs (when compared to manual instrumentation) due to over-instrumentation [27, 49, 149]. Basically, the compiler cannot precisely determine which instructions indeed access shared data and hence has to instrument the code conservatively. This results in unnecessary calls to STM functions, reducing the performance of the resulting code.

3. **Transparent privatization.** Making certain shared data private to a certain thread is known as *privatization*. Privatization is typically used to allow non-transactional accesses to some data, either to improve performance by avoiding costs of STM calls when accessing private data or to support legacy code. Using privatization with unmodified STM algorithms (that use invisible reads) can result in various race conditions [135]. There are two different approaches to avoiding such race conditions: (1) a programmer marks transactions that privatize data, so the STM can safely privatize data only for these transactions or (2) the STM ensures that all transactions safely privatize data. We call the first approach *explicit* and the second *transparent*. Explicit privatization places additional burden on the programmer, while transparent privatization incurs sometimes

Benchmark suite	Workload description
STAMP	bayes: bayesian networks structure learning genome: gene sequencing intruder: network intrusion detection kmeans: partition-based clustering labyrinth: shortest-distance maze routing ssca2: efficient graph construction vacation: travel reservation system emulation yada: Delaunay mesh refinement
Synchrobench	hash table implementation of a set sorted linked list implementation of a set skip list implementation of a set red-black tree implementation of a set
STMBench7	read-dominated large synthetic benchmark with 10% write operations read/write large synthetic benchmark with 60% write operations write-dominated large synthetic benchmark with 90% write operations

Table 2: Benchmarks used for evaluating Software Transactional Memory on multicores

high runtime overheads [149]. In particular, with explicit privatization no transaction pays any additional cost if it does not use privatization, while with transparent privatization all transactions are impacted. This cost can be high, especially in cases when no data is actually being privatized.

2.2.1 Leveraging the concurrency of multicore platforms

We now present our methodology to re-evaluate the performance of software transactional memory on top of multicore platforms. Our goal was to compare STM performance to sequential code using (i) a larger set of benchmarks and (ii) a real hardware that supports higher levels of concurrency than the previous study [28]. To this end, we experimented all ten workloads of the STAMP (v0.9.10) benchmark suite [26], four workloads of an early version of the C/C++ Synchrobench benchmark suite [59], and three different STMBench7 [74] workloads to evaluate STM performance on both large and small scale workloads, as summarized in Table 2. We also considered two hardware platforms—the UltraSPARC T2 CPU machine and the 4 quad-core AMD Opteron x86 CPU machine we presented in Table 1.

Finally, we evaluated a state-of-the-art STM algorithm, SwissTM [47]. but also executed experiments with TL2 [40], McRT-STM [127] and TinySTM [53] to confirm our observations. In addition, we were provided with the Bartok STM [1, 82] performance results on a subset of STAMP by Tim Harris at that time at Microsoft Research. SwissTM [47] is a word-based and time-based STM that uses a variant of two-phase locking for concurrency control and invisible reads, similar to TL2 [40] and TinySTM [53]. It detects write/write conflicts eagerly to abort rapidly a transactions that is likely doomed to abort but detects read/write conflicts lazily to limit unnecessary aborts. It also uses a shared counter for contention management similar to Greedy [71] to aborts transactions that performed less work but avoids short transactions to update this counter.

We implemented privatization support in SwissTM using a simple validation barriers scheme described in [109]. To ensure safe privatization, each thread, after committing a transaction t_0, waits for all other concurrent transactions to detect the changes to shared data performed by t_0. Basically, the thread waits for all concurrent threads to commit, abort or validate before executing application code after the transaction. None of the benchmarks we use requires privatization, which means that we measure the worst case: namely, supporting transparent privatization only incurs overheads, without

STM Speedup		Manual instr. Explicit priv.			Compiler instr. Explicit priv.			Manual instr. Transparent priv.			Compiler instr. Transparent priv.		
Hardware	Hw threads	avg	min	max	avg	min	max	avg	min	max	avg	min	max
Sun SPARC	64	9.1	1.4	29.7	-	-	-	5.6	1.2	23.6	-	-	-
Intel x86	16	3.4	0.54	9.4	3.1	0.8	9.3	1.8	0.34	5.2	1.7	0.5	5.3

Table 3: Summary of the STM speedups over sequential code on multicore architectures

the performance benefits of reading and writing privatized data outside of transactions as in [131]. We used Intel's C/C++ STM compiler [120, 127] for generating compiler instrumented benchmarks, however, Intel's C/C++ STM compiler only generates x86 code thus we were not able to use it for our experiments on SPARC. The compiler simplifies the job of a programmer who only has to mark atomic blocks of code. To be exhaustive we consider all combinations of privatization and compiler support for STM.

All of these experiments confirm our general conclusions that STM performs well on a wide range of workloads. More precisely, our results, summarized in Table 3, show that STM with manual instrumentation and explicitly privatization outperforms sequential code in all the benchmarks on both hardware configurations, except high contention write-dominated STMBench7 workload on x86; by up to 29 times on SPARC with 64 concurrent threads and by up to 9 times on x86 with 16 concurrent threads.

Compiler over-instrumentation does reduce STM performance, but does not impact its scalability. With such an instrumentation and explicit privation, STM outperforms sequential code in all STAMP benchmarks with high contention and in all but one micro-benchmark; by up to 9 times with 16 concurrent threads on x86. Support for transparent privatization impacts STM scalability and performance more significantly, but with manual instrumentation and transparent privation, STM still outperforms sequential code in all benchmarks on SPARC and in all but two high contention STMBench7 workloads, one high contention STAMP benchmark and one micro-benchmark on x86. STM with manual instrumentation and transparent privatizations outperforms sequential code by up to 23 times on SPARC with 64 concurrent threads and 5 times on x86 with 16 threads. Even when both transparent privatization and compiler instrumentation are used (and even though we used relatively standard techniques for both), STM still performs well, outperforming sequential code in all but two high contention STAMP benchmarks and one micro-benchmark (3 out of 14 workloads), by up to 5 times with 16 concurrent threads on x86.

2.2.2 On the reasons why STM can be more than a research toy

We believe that the reasons for such considerable difference between STM performance in our experiments and in the previous study [28] are three-fold:

1. *Workload characteristics.* After getting the details of experimental settings of [28] from the authors, we noticed that their experiments did not use the default STAMP workloads. Configurations used in [28] significantly increase the contention in all three STAMP benchmarks, by reducing the size of the shared data in `kmeans` and `vacation` and reducing the fraction of read-only transactions in `genome`. Speculative approaches to concurrency, such as STM, do not perform well with very high contention workloads, which is confirmed by our experiments, in which STM has the lowest performance in very high contention benchmarks (e.g., `kmeans high` and `STMBench7 write`). By contrast, to evaluate the impacts of workload characteristics on the performance, we ran STAMP workloads from [28].

2. *Different hardware.* We used hardware configurations with the support for more hardware threads—64 and 16 hardware threads in our experiments compared to 8 in [28]. This lets STM perform

better as there is more parallelism at the hardware level to be exploited. Also, our x86 machine does not use hyper-threading, a feature that consists of running multiple hardware threads on a single Intel core, while the one used in [28] uses a quad-core hyper-threaded CPU. Hardware thread multiplexing in hyper-threaded CPUs can degrade performance when compared to the machine supporting the same number of hardware threads without multiplexing.

3. *More efficient STM.* We believe that part of the performance difference comes from a more efficient STM implementation. The results of [47] suggest that SwissTM has better performance than TL2, which has comparable performance to the IBM STM used in [28].

To summarize, our experiments show that STM indeed outperforms sequential code in most configurations and benchmarks, offering already now a viable paradigm for concurrent programming. These results are important as they support initial hopes about the good performance of STM, and motivate further research in the field.

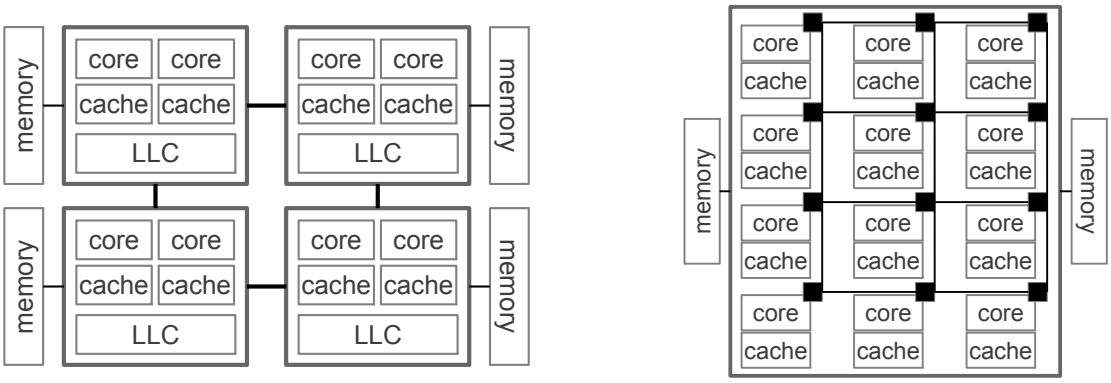
In its software form, called STM [133], the transaction paradigm can be implemented without requiring any specific hardware support: at least in principle. In practice, this is not entirely true because STMs do typically assume multicore architectures and rely on an underlying cache-coherent system.

2.3 Leveraging Manycores [64]

Recently, manufacturers started producing manycore processors, with the idea of increasing the number of cores placed on a single die while decreasing their complexity for enhanced energy consumption [19, 138]. Contemporary manycores comprise a range from few to a thousand cores [121]. This scale shift motivated manufacturers to experiment with processors without cache coherence [92] and requires now new programming techniques [30, 97]. The main reason to radically rethink or avoid cache coherence is that it incurs core-to-core traffic [111] that gets amplified when multiple cores synchronize [23, 38]. This traffic boils down to performance drops in existing systems so that they require either drastic synchronization changes [22, 51] or synchronization via message passing instead of shared memory [14] for their performance to scale up even sometimes to only 32 cores. As existing STMs build on top of shared memory and rely on the underlying cache coherence, they cannot be ported on these non-coherent platforms. In this section, we present our solution, called TM^{2}C, by presenting an extended version of our original work [64].

2.3.1 A transactional memory for manycore platforms

We proposed a novel transactional-memory protocol, called TM^{2}C, standing for *Transactional Memory for Many-Cores* [64]. TM^{2}C provides a simple transactional interface to the programmer of concurrent applications and does not require a coherent memory interface from the architecture. TM^{2}C is the first TM system tailored for non-coherent many-core processors. TM^{2}C capitalizes the low latency of on-die message passing and the absence of coherence overhead by exploiting visible reads and allowing the detection of conflicts whenever a transaction attempts to override some data read by another transaction. Thus, TM^{2}C anticipates the conflict resolution, otherwise deferred to the commit phase of the reading transaction. TM^{2}C can be seen as a distributed transactional memory (DTM), a paradigm often used on top of real networks, but in contrast to the network-on-chip of manycores, high latency networks typically require to pipeline asynchronous reads (inherently invisible) to achieve reasonable performance. TM^{2}C is weakly atomic [110] in that the transactionally accessed data should not be concurrently accessed by non-transactional code.



(a) Four interconnected symmetric multi-processors (SMPs)

(b) A chip multi-processor (CMP)

Figure 2: High-level architecture of (a) multicore and (b) manycore

Internals. First, we implemented a message-passing library for the communication in TM^{2}C. This library allocates two cache lines for every two-node combination $\langle i, j \rangle$. One is used as a buffer for x sending messages to j while the other is used for the messages of j to i. The library does not use atomic operations: the cores (or *nodes*) synchronize by spinning on a flag (one word of the cache-line buffer). To send a message, a node checks the flag to see whether the buffer is free and when it is free, the node writes the message and updates the flag to indicate that the buffer contains a message.

A transactional request, such as a `read`, corresponds to a library-based message to the node responsible for this address. This message includes the opcode of the request (e.g., `TX_READ`), some contention management metadata, and the transactional memory address that the request wants to access. The shared memory used by the TM can either be a memory region of the non-coherent shared memory that the hardware provides, or a partitioned global address space (PGAS). In the case of non-coherent shared memory, which is the default one, the DTM protects the shared memory by granting read or write locks to transactional requests before they can proceed accessing the address. In the case of PGAS, each DTM node, apart from being responsible of protecting a partition of the memory, also keeps the only copy of the actual memory in its local memory. A non-conflicting transactional read includes the actual data in the response. Similarly, a transactional write includes the data to be written in the request message. Upon a successful commit of the transaction, the DTM nodes write these data, that have been buffered on the server side, to the PGAS memory.

Reads. Transactional reads work with early-lock acquisition and therefore the system operates with visible reads. Early acquisition suggests that a transaction has to acquire the corresponding read-lock before proceeding to the actual read. The visibility of reads is an outcome of the early acquisition: every transaction is able to "see" the reads of the others due to the read-locks. The motivation behind this design decision is twofold. First, manycores provide fast message-passing mechanisms and tend to limit the cache-coherence support. Taking this into account, the overhead of performing synchronous read validation is acceptable. On a cluster, the higher messaging latency would make a synchronous solution prohibitive. Second, visible reads are often problematic for affecting the cache behavior of the system in traditional TM [36] because reading requires to acquire a lock which might translate into additional cache misses [23]. In TM^{2}C this is not the case due to the use of message passing rather than cache

coherence. The visibility of reads does not require changes to local memory objects (e.g., locks), instead it uses a distributed lock service to acquire the corresponding locks by communicating with a remote node. Additionally, visible reads are necessary for starvation-free contention management. Without the read visibility, the write-after-read conflict detection is deferred to a validation phase typically just before the commit. If a conflict is then detected, it is too late to perform conflict resolution because the writing transaction has already committed the new values, i.e., it made the writes visible to other transactions. As a result, the reader would be aborted, potentially leading to starvation.

Writes. Transactional write is the operation used to write to a memory object within the context of a transaction. TM^{2}C implements writes with lazy-lock acquisition and deferred writes[10]. Every write operation is buffered and the locks are actually acquired once the commit phase is reached. We chose lazy-lock acquisition for two reasons. If two transactions conflict, one has to write on the memory object. Therefore, if a transaction holds a write-lock for a long time, it increases the probability that a (read-after-write or write-after-write conflict might appear. Lazy write acquisition reduces the time that the write-locks are being held. Moreover, deferred writes allow write-lock batching: requesting the locks for multiple memory objects in a single message, which can significantly reduce the number of messages and improve performance.

DTM servers. The DTM servers receive requests as library messages. The servers keep the transactional metadata (i.e., the read/write-locks) in a sequential hash table maintained by each DTM: read/write-lock entries are inserted using the requested address as the key. A transactional request from node R, for address K, on node N proceeds with the following steps:

1. N accesses the hash table. If there is no existing entry for address K, an entry is inserted with the requested access rights provided to R.
2. Otherwise, N checks whether the request of R generates a conflict with the existing access rights. If it does not[11], the entry of K is updated to reflect the access rights given to R.
3. Otherwise, N uses the contention manager (CM) to decide whether the transaction of R, or the existing transaction(s) have to be aborted. If the CM decides to abort R, then the entry in the hash table remains unchanged.
4. Otherwise, after the existing transaction(s) have been aborted, the entry is updated with the new access rights.

Using a dynamic data structure instead of a fixed array of metadata has an important benefit. TM^{2}C is able to support byte-level granularity for conflict detection, because the size of the hash-table structure does not depend on the granularity, but only on the number of entries that have been inserted. Accordingly there is no overhead if the granularity is a byte, a cache line, or a whole page.

Dedicated resources vs. multitasking with run-to-completion. In TM^{2}C, the application and the distributed transactional memory (DTM) services of TM^{2}C are independent: the application executes transactions by requesting data accesses, while the DTM decides whether an access can be granted. Both services are fully distributed and could either both be deployed on the same cores or nodes, all exploiting each node but at different time slots, or deployed on distinct nodes, concurrently exploiting different nodes. The former deployment strategy thus leads to multitasking while the latter leads to dedicating cores to services.

[10] The strategy of deferring writes is also known as write-back.

[11] The only such case is when K is being read and the new request is a read as well.

Our initial design used multitasking to allow both the application and the DTM system to run on every node. A user-space library, called libtask[12], was used for this implementation. We preferred libtask over POSIX threads because it offers significantly cheaper context switches. Libtask is a simple coroutine library that gives the programmer the illusion of threads, but the operating system sees only a single kernel thread. The multitasking approach suffers however from an important limitation: the scheduling of node j can potentially affect the execution of node i, where $j \neq i$. Node j is executing some application code while node i tries to complete a service request that involves node j. The request is not served until j completes its application computation. As a result, there is an additional delay that increases the latency of the service operation.

As a manycore provides a large number of simple cores, assigning a dedicated role to each core better exploits parallelism. As a follow-up to this observation, we engineered a second deployment strategy in which disjoint sets of cores are dedicated to hosting distinct services. This solution overcomes the issues of multitasking by avoiding timing dependencies between the application and the DTM. The nodes dedicated to the same service (e.g., DTM) do not need to communicate with each other. This leads to a complete decoupling among the DTM and the application cores, respectively.

2.3.2 Managing contention in a distributed environment

Implementing contention management on top of a manycore is a difficult problem because it is hard to guarantee that every transaction will eventually commit if the memory is not coherent. Existing CMs, designed for multicores, are generally centralized [71,128,129] and are not applicable to message passing as they either rely on a global counter (e.g., Greedy, PublishedTimestamp, Timestamp), randomization (e.g., KinderGarten), or on constantly changing priorities that become inaccurate when conflict resolution gets propagated in an asynchronous system (e.g., Eruption, Karma, Polka).

Take the contention manager Greedy that we choose for multicore (§§§2.2.1) as an example. Greedy prioritizes transactions using timestamps, which represent the times at which transactions started. Upon a conflict, the youngest conflicting transactions are aborted in favor of the oldest one to guarantee that every transaction commits eventually, possibly after having aborted several times. In a distributed system, the lack of a global clock prevents us from implementing Greedy as different nodes of the system do not have a way of taking consistent timestamps. Typically, the transaction with the most advanced clock might obtain a lower priority even though it starts first.

Now, consider a message passing variant of Greedy, namely *Offset-Greedy*, that estimates timestamps based on time offsets. When a transaction starts, it acquires a timestamp t_{start} using the local clock of the node on which the transaction executes. Offset-Greedy then takes the following steps whenever a node performs a transactional operation:

1. The transaction takes a new local timestamp $t_{current}$ and calculates the time offset since the beginning of the transaction ($t_{offset} = t_{current} - t_{start}$).
2. The transaction sends the request to the responsible core running the distributed transactional memory algorithm, piggybacking t_{offset}.
3. The distributed transactional memory algorithm takes a local timestamp $t'_{current}$ and uses the t_{offset} from the request to estimate the timestamp of the transaction according to its own local clock ($t_{estimate} = t'_{current} - t_{offset}$).
4. The request is normally processed.

In the presence of conflicts, Offset-Greedy uses the estimated timestamps to order transactions (as Greedy does). However, Offset-Greedy does not guarantee starvation-freedom. The offset calculation

[12] http://swtch.com/libtask/.

technique does not take into account the message delays in computing the offset. As the DTM load impacts the message delay, nodes can obtain inconsistent views of timestamps. As a result, if a conflict emerges concurrently on two DTM nodes with inconsistent views, both transactions might be aborted by Offset-Greedy. This scenario could lead to livelocks where transactions repeatedly abort each other.

We propose a distributed contention manager, FairCM, that guarantees that every transaction eventually commits. Visible reads allow us to use contention management in a way similar to STMs, yet fully decentralized, to guarantee that all transactions eventually commit.

FairCM is fair regarding the effective transactional time of each node. FairCM prioritizes the transactions using the cumulative time spent on successful transaction attempts. Upon conflict, the node that has spent most time is aborted. If two nodes have the same number of committed transactions, then their identifiers are used as tie-breakers. So, if a transaction proceeds as follows:

$$\text{Start} \rightarrow \text{Abort}_1 \rightarrow \text{Restart}_1 \rightarrow \text{Abort}_2 \rightarrow \textbf{Restart}_2 \rightarrow \textbf{Commit}$$

then only the duration from **Restart**$_2$ to **Commit** will be added to the cumulative time. Upon a conflict, the transaction on the node with the lowest cumulative time has the highest priority. FairCM guarantees that every transaction eventually commits. Note that in the unlikely case where two nodes have the same effective transactional time, the nodes' ids are used as tie-breakers to deterministically resolve the conflict.

Evaluation on multicores and manycores. We experiment with TM^{2}C using a hash table and a linked list benchmarks from an early version of C/C++ Synchrobench [59] as well as on a bank and a MapReduce synthetic applications. The bank application uses transactions for transferring and computing the balance of bank accounts. Bank was first used to evaluate shared memory STMs [87]. It features potentially long read-only transactions summing the amount of all 1024 bank accounts and short update transactions accessing only a couple of bank accounts to transfer amounts. We designed the MapReduce application to take a text file as the input and counts the number of occurrences of each letter. TM^{2}C takes the role of allocating chunks of the file to cores and of updating the total statistics atomically, thus removing the need for a centralized master node.

We used three architectures for our experiments, two manycore and one multicore architectures: the 48-core Intel SCC, the 36-core Tilera TileGx and the 48-core AMD Opteron depicted in Table 1. We observed that the number of nodes that must be dedicated to the DTM service heavily depends on the considered workload. We evaluated the performance variation of the bank application. The peak performance on the TileGx occurs around 12 service cores whereas the SCC and the Opteron seem to have optimal performance around 16 cores. Interestingly, these numbers represent $\frac{1}{3}$ of the total number of cores present on each machine: 12 out of 36 on the TileGx and 16 out of 48 on the SCC and the Opteron. As a result of these observations, we set TM^{2}C to dedicate by default twice as many cores to the DTM service as to the application service.

To capture the overall benefits of writing concurrent programs with TM^{2}C on many-cores, we compare the performance of the transactional code against the performance of the bare sequential code on the 48-core SCC. For the MapReduce benchmark we dedicated only one core to host the DTM service while the remaining 47 cores executed the application. In this case, TM^{2}C outperforms sequential code by up to 39 times for 4 KB chunks on an operating system with a 4 KB-sized page. On the Synchrobench hash table benchmark varying from 20% to 50% updates, the transactional implementation outperforms sequential code by 30 times. Moreover, as writes to the shared memory are more expensive than reads, we observed that a higher update ratio would lead to lower performance but the performance drop of the sequential version using a single memory controller is more important than TM^{2}C that leverages all memory controllers, thus mitigating the penalties induced by the write operations.

We also compared TM^{2}C on the SCC, the TileGx and the Opteron using the bank application, the hash table, and the linked-list benchmarks. Overall, all results closely follow the trends of the latencies

of message passing we observed on these platforms. In other words, the TileGx results outperform the Opteron results that outperform, in turn, the SCC results. When comparing a centralized variant of TM^{2}C that exploits multicore cache coherence, we could observe that the performance on AMD Opteron would be higher than the one on Tilera TileGx. We also observed using these two TM^{2}C variants that, as expected, a traditional TM implementation seems more suitable for cache-coherent multicore platforms like the AMD Opteron whereas a DTM seems more suitable for manycore platforms like the Tilera TileGx, as it provides similar performance under normal contention and significantly better performance and scalability under high contention.

To conclude, even though we did not use any compiler support for distributed transactional memory, TM^{2}C demonstrates that TM can provide a simple transactional interface to programmers, guaranteeing the eventual successful termination of all their transactions, while hiding the complex architectural communication features, like caches, off-chip interconnect, or network-on-chip. In addition, a TM can leverage the message passing latency of the underlying platform to outperform the bare sequential code (running on a single core) and to scale with the number of cores. Its portability allows it to be deployed on non-cache-coherent machines with limited support of atomic operations (e.g., test-and-set). Its performance can scale with the level of concurrency of manycore machines to significantly outperform sequential performance.

While the development of a transactional memory can be quite intricate depending on the underlying support (cache-coherence, hardware message passing), when it is properly developed, it simplifies the task of writing a concurrent program by simply requiring the programmer to write sequential code and identify regions of code that should execute atomically, the transactions.

2.4 Conclusion

TM appeared in the 90's, started becoming a technology following the hype cycle when multicores were announced by manufacturers around 2003, and seems to have reached maturity today. The advent of multicores served as a technology trigger for TM to get rapid momentum from 2003, when the first dynamic STM [88] was proposed, and to 2008, when it seemingly reached a peak of inflated expectations. During that period the first edition of a book on the topic was published [79], the first workshop on the topic, the *ACM SIGPLAN Workshop on Transactional Computing*[13], started, a major correctness criterion for TM was defined [72] and the first hybrid TMs combining hardware and software were proposed [37,102]. From 2008 till 2010, skepticism appeared in a through of disillusionment, first when researchers called STM a "research toy" [27], then with the cancellation of the manufacturing project of the Sun Rock processor with HTM support and finally with the cancellation of the Microsoft STM.NET project aiming at supporting STM in .NET.

Since 2010, we have followed a phase of enlightenment with the first transactional memory stack [56], the TM support in gcc v4.7 in 2012, new performance evaluations showing that STM could, afterall, leverage multicores [46] and the appearance of Hardware Transactional Memory in the IBM BlueGene/Q [142]. It seems now that we have reached a plateau of productivity, as transactional memory is available to the large public through the Intel Haswell and the IBM Power processors [116]. Although transactional memory is now a well-established programming paradigm, it is not a panacea. In particular, we illustrate in the sequel the concurrency limitation inherent to transactional memory (§3) and the solution we proposed to address it (§4).

[13] http://transact2015.cse.lehigh.edu/.

3. Identifying Concurrency Limitations

In this section we show that in their classic form, transactions prevent us from extracting the same level of concurrency possible with more primitive synchronization techniques on some workloads. Not surprisingly, researchers have been exploring relaxations of the classic transaction model [106, 118, 124] that enable more concurrency. Nevertheless, it proved challenging to do so while keeping the simplicity of the original model, namely, its ability to (i) preserve the original sequential code and (ii) compose applications devised by different programmers, possibly with different skills. Perhaps more surprisingly, we show here that this limitation is actually inherent to the transaction concept but not to the way it is used.

3.1 Measuring Schedule Acceptance [65]

We distinguish the notions of type and its implementation that relies typically on a separate data structure. We then define the concurrency metric of an implementation as the amount of schedules it accepts. This definition, inspired by the analysis of parallelism in database concurrency control [84, 148] is the key to compare the concurrency obtained when synchronizing the same sequential code with different techniques. This work extends our previous notion of input acceptance, originally designed for transactional memory [65], and reuses some of the notions we introduced in [66].

3.1.1 Preliminaries

System. We assume an asynchronous shared-memory system in which *n processes* $p_1, \ldots, p_n$, where $n \in \mathbb{N}$, communicate by invoking operations on objects of some abstract type so that each operation executes a sequence of accesses on shared *base objects* before returning their response [85]. An *access*, $a \in \Phi$ on a base object x is either a read $r(x)$ that returns the value of x, a write $w(x, *)$ that returns `ok` or a read-modify-write $rmw(x, *, *)$ that returns `true` or `false`.

Types. A *type* τ is a tuple $\langle \Pi, \Gamma, Q, \iota, \Delta \rangle$ where Π is the set of possible operations, Γ is the set of possible returned values, Q is a set of states, $\iota \in Q$ is an initial state and $\Delta \subseteq Q \times \Pi \times Q \times \Gamma$ is a transition that given a state and an operation determines the possible operation responses and resulting states. A τ'-set type is a type $\mathcal{S}$ with operations $\Pi = \{\texttt{insert}(e \in \tau'), \texttt{delete}(e \in \tau'), \texttt{contains}(e \in \tau')\}$ that apply to the states $Q = (\tau')^*$ of a set of object of some given type τ'. (We use the subscript τ to refer to the sets Π, Γ, Q, Δ and initial state ι of the specific type τ, e.g., Π_τ) The *sequential specification* of a set type $\mathcal{S}$ is defined by its transitions $\{q, \pi, q', r\} \in \Delta$ as depicted in Table 4. Later we will present other types like a snapshot type (§§§3.3.2) and a dictionary type (§§§4.3.2).

π	`insert`(e)	`delete`(e)	`contains`(e)
$e \in q$	$r = \texttt{false} \wedge q' = q$	$r = \texttt{true} \wedge q' = q \setminus \{e\}$	$r = \texttt{true} \wedge q' = q$
$e \notin q$	$r = \texttt{true} \wedge q' = q \cup \{e\}$	$r = \texttt{false} \wedge q' = q$	$r = \texttt{false} \wedge q' = q$

Table 4: The sequential specification of the τ'-set type with q the current state, $\pi = \{\texttt{insert}(e), \texttt{delete}(e), \texttt{contains}(e)\}$, $e \in \tau'$, r the returned value and q' the resulting state

Implementations. A *sequential implementation* $\mathbb{S}_\tau$ of a type τ is a mapping from Π_τ to a set of possible sequences of accesses such that for all $\pi \in \Pi_\tau$ we have $\mathbb{S}_\tau(\pi) = \{\langle \sigma, \prec_\sigma \rangle\}$. The reason why each

operation π is not mapped to a single sequence is because the sequence of accesses depends on the state of the type when the operation executes. A *concurrent implementation* $\mathbb{C}_\tau$ of a sequential implementation $\mathbb{S}_\tau$ of type τ is a mapping $\mathbb{S}(\Pi_\tau) \mapsto \mathbb{C}(\mathbb{S}(\Pi_\tau))$ where for all $seq = \langle \sigma, \prec_\sigma \rangle \in \mathbb{S}(\Pi_\tau)$, $\mathbb{C}_\tau(seq) = \langle \gamma, \prec_\gamma \rangle$ such that $\sigma \subseteq \gamma$ and $\prec_\sigma \subseteq \prec_\gamma$. Intuitively, function $\mathbb{C}_\tau$ has a set of accesses $acc(\mathbb{C}_\tau) = \gamma$ that consists of accesses α and other accesses $\gamma \setminus \alpha$ to metadata based objects used to prevent data races upon execution of the concurrent implementation. During an execution of a concurrent implementation $\mathbb{C}_\tau$, an access in σ returns a value in $ret(\mathbb{C}_\tau)$, a metadata access in $\gamma \setminus \sigma$ returns a value in $ret(\mathbb{C}_\tau) \cup \{\perp\}$ and an operation returns value in $\Gamma_\tau \cup \{\perp\}$ because both metadata accesses and operations can return a special value $\perp$ indicating that the access or the operation *aborts*.

3.1.2 Schedules and concurrency

We consider three types of *events*, the invocations $inv(\Pi_\tau)$ of operations Π_τ as well as their responses coupled with their corresponding returned values, $resp(\Pi_\tau) \times (\Gamma_\tau \cup \{\perp\})$, and the base object accesses of these operations coupled with their returned values $acc(\mathbb{C}_\tau) \times (ret(\mathbb{C}_\tau) \cup \{\perp\})$. We assume that all accesses execute instantaneously, hence we do not distinguish their invocation from their response and two accesses cannot be concurrent.

Executions. An *execution* α is a sequence of events $\alpha = \langle E, \prec_\alpha \rangle$. We assume that executions are *well-formed*: (i) no process invokes a new operation (resp. access) before the previous operation (resp. access) returns, and (ii) no operation returns before all its accesses return or some of its accesses returned $\perp$, in which case the operation returns $\perp$.

Let $\alpha|p_i$ (resp. $\alpha|\Pi$) denote the subsequence of an execution α restricted to the events of process p_i (resp. events of operations $\pi \in \Pi$). An operation is *complete* in α if its invocation event is followed by a response event. Execution α is *complete* if every operation is complete in α. The committed execution of α is a complete execution $\alpha' = committed(\alpha)$ such that for any operation $\pi \in \alpha$, the returned value of π is $ret(\pi) \neq \perp$ (or equivalently for any event $a \in \alpha$, its returned value is $ret(a) \neq \perp$).

Correctness. Executions α and α' are *equivalent* if for every process p_i, $\alpha|p_i = \alpha'|p_i$. An operation π *precedes* another operation π' in an execution α, denoted $\pi \rightarrow_\alpha \pi'$, if the response of π occurs before the invocation of π'. Two operations are *concurrent* if neither precedes the other. An execution is *sequential* if it has no concurrent operations. A sequential execution α is *legal* if for every object x, every read of x in α returns the latest written value of x.

Let a *history* be a committed execution. A complete history H is *linearizable* with respect to an object type τ if there exists a sequential high-level history S equivalent to H such that (i) $\rightarrow_H \subseteq \rightarrow_S$ and (ii) S is consistent with the sequential specification of type τ. A history H is linearizable if it can be *completed* (by adding matching responses to a subset of incomplete operations in H and removing the rest) to a linearizable history [12, 91]. We only consider correct implementations, hence all the histories of a concurrent implementation we consider are linearizable with respect to the type τ.

Schedules. The *schedule* $\sigma = \langle E_\sigma, \prec_\sigma \rangle$ is *exported* by an execution α if it is the sequence of the committed execution $\alpha' = committed(\alpha) = \langle E_{\alpha'}, \prec_{\alpha'} \rangle$ without returned values. More formally, we define a projection p from all events $E_{\alpha'}$ to the events of E_σ such that if $e \in E_{\alpha'}$, (i) either e is an invocation and $p(e) = e \in E_\sigma$ or (ii) $e = \langle r, v \rangle$ is a response/value pair and $p(e) = r \in E_\sigma$ but $v \notin E_\sigma$ and the order of events is preserved under the projection: for all pairs of events $e, e' \in E_{\alpha'}$ we have $e \prec_{\alpha'} e'$ if and only if $p(e) \prec_\sigma p(e')$.

Concurrency. We say that an implementation *accepts* a schedule σ, if it contains a complete execution α that exports schedule σ. Otherwise we say that the implementation *rejects* the schedule σ. We measure

the amount of concurrency provided by a concurrent implementation $\mathbb{C}$ as the set of schedules it accepts. More precisely, we say that an implementation $\mathbb{C}$ *allows for more (resp. strictly more) concurrency* than another implementation $\mathbb{C}'$ if the amount of schedules exported by the executions of $\mathbb{C}$ is larger (resp. strictly larger) than the amount of schedules exported by the executions of $\mathbb{C}'$.

Our concurrency definition helps comparing multiple implementations of the same data type. It builds upon the notion of input acceptance, in that the concurrency measures the variety of schedules imposed by the environment that an implementation accepts. Note that concurrency differs from permissiveness [70]. In fact, a multiversion transactional memory achieves high permissiveness and low concurrency if it accepts a single schedule but produces many histories by randomly choosing return values among the correct ones. Based on our concurrency metric, we can now compare a transaction-based implementation and a lock-based implementation of a list-based set.

3.2 Rejected Concurrency Schedules [60]

In this section, we show the concurrency limitation of transactions by comparing them to fine-grained locks. We propose an example of a schedule that is accepted by a lock-based implementation but cannot be accepted by any transactional memory algorithm.

Algorithm 1 Sequential and concurrent (transactional and lock-based) implementations of `contains`

```
1:  seq-contains(val)_p:
2:      int result;
3:      node *prev, *next;
4:      curr = set→head;
5:      next = curr→next;
6:      while next→val < val do
7:          curr = next;
8:          next = curr→next;
9:      result = (next→val == val);
10:     return result;

11: tx-contains(val)_p:
12:     int result;
13:     node *prev, *next;
14:     curr = set→head;
15:     transaction {
16:         next = curr→next;
17:         while next→val < val do
18:             curr = next;
19:             next = curr→next;
20:     result = (next→val == val);
21:     }
22:     return result;

23: lk-contains(val)_p:
24:     int result;
25:     node_lk *prev, *next;
26:     lock(&set→head→lock);
27:     curr = set→head;
28:     lock(&curr→next→lock);
29:     next = curr→next;
30:     while next→val < val do
31:         unlock(&curr→lock);
32:         curr = next;
33:         lock(&next→next→lock);
34:         next = curr→next;
35:     unlock(&curr→lock);
36:     result = (next→val == val);
37:     unlock(&next→lock);
38:     return result;
```

3.2.1 Using transactions or locks to synchronize a list-based set

To make our point that the expressiveness of transaction is limited we consider a set type τ implemented with a sorted linked list data structure, also called a list-based set, and more precisely its `contains` $\in \Pi_\tau$ operation. Consider Algorithm 1 that depicts a `contains` operation of a sequential implementation (`seq-contains` operation on the left) and two concurrent implementations: one based on transaction (`tx-contains` in the middle), the other based on locks (`lk-contains` on the right). We omit the pseudocode of operations `delete` and `insert` of each implementation but assume that each sequential operation is not synchronised, each transactional operation is simply enclosed in a transaction and each lock-based operation uses hand-over-hand locking or lock-coupling [16] as described

elsewhere [90, Ch.9].

process p_1	process p_2	process p_3
inv(`contains`(4))		
$r(h)$		
$r(x)$		
	inv(`insert`(1))	
	$r(h)$	
	$r(x)$	
	$w(h)$	
	resp(`insert`)	
		inv(`insert`(4))
		$r(h)$
		$r(n)$
		$r(x)$
		$r(z)$
		$r(t)$
		$w(z)$
		resp(`insert`)
$r(z)$		
$r(m)$		
resp(`contains`)		

Table 5: A typical list-based set schedule where three processes execute `contains`(4), `insert`(1) and `insert`(4), respectively, on a set type with initial state {2,3}; this schedule is rejected by transaction-based concurrent implementations $\mathbb{C}_\tau$ whereas it can be accepted by a lock-based concurrent implementation $\mathbb{C}'_\tau$

A transaction delimits a region of accesses to shared locations and protects the set of locations that is accessed in this region. By contrast, a (fine-grained) lock generally protects a single location even though it is held during a series of accesses. This is a crucial difference between transactions and locks in terms of expressiveness and concurrency. Table 5 depicts a hypothetical schedule exported by the execution of a concurrent list-based set implementation, where time increases from top to bottom. This schedule could result from a sorted linked list implementation of an integer set type with initial state $\{2, 3\}$ and processes p_1, p_2 and p_3, executing a `contains`(4), an `insert`(1) and an `insert`(4), respectively, of a concurrent implementation $\mathbb{C}_\tau$. (We omitted some linked list accesses in Figure 1 for the sake of clarify in the presentation.)

Process p_1 starts first by reading the two first nodes of the sorted linked list, the head h and the node x with value 2 and keeps the address of the tail node t. Now, process p_1 gets preempted and process p_2 reads the two first nodes, h and x, before realizing that value 1 should be inserted between these two nodes. Then, p_2 allocates a new node n with value 1 whose successor is set to node x with value 2. These are private steps of p_2 not depicted here for simplicity as the new node n is not reachable from the linked list yet. The new node n becomes reachable only after p_2 writes h for the head's successor to points to n. After p_2's insertion returns, process p_3 traverses the list by reading all the nodes, including the new node n, until the tail node t, and inserts a new node, say m, with value 4 between z and t. Finally, process p_1 reads the value of z at the address it stored previously, reads m and t, and returns as soon as it observes that it reached the end of the list.

3.2.2 Schedule acceptance depending on synchronization

While Table 5 shows only a schedule without depicting the input or returned values, we can easily observe that the two insertions successfully added new nodes and the contains succeeded in finding value 4, hence they should all return `true`. We can deduce also that at the end of this execution the data structure is a sorted list of nodes h, n, x, z, m, t. Given the values we assumed, the final state of the set this list implements is $\{1, 2, 3, 4\}$. We can thus represents a history (where we omit returned value) as:

$$\mathcal{H} = r(h)^1, r(x)^1, r(h)^2, r(x)^2, w(h)^2, r(h)^3, r(n)^3, r(x)^3, r(z)^3, r(t)^3, w(z)^3, r(z)^1, r(m)^1.$$

Finally, assuming that every read on a location returns the last written value at that location, we can conclude that the history is linearizable as there exists an equivalent sequential history $\mathcal{S}$ that respects the sequential specification of the set type where the `insert`$(1)^2$ returns `true`, then the `insert`$(4)^3$ returns `true` and finally the `contains`$(4)^1$ returns `true` as well.

Interestingly, the schedule exported by the execution of $\mathcal{H}$ cannot be accepted with a classic (opaque) transactional memory. Consider the `contains` operation of the transaction-based concurrent implementation $\mathbb{C}_\tau$ depicted in Figure 1 and let π_1 denote the transactional `contains` of process p_1, and let π_2 and π_3 denote the transactional `insert` of p_2 and p_3, respectively. Operation π_1 starts before reading the next pointer of the head node. Hence when the next pointer of the head node gets updated by π_2, then a conflict is detected between the two transactions of the two first processes. Finally, π_3 writes the next pointer of z that gets then read by π_1, hence there is another conflict that should be detected between π_3 and π_1. The two conflicts indicate that in a possible equivalent sequential history S we would need $\pi_1 \rightarrow_S \pi_2$ and $\pi_3 \rightarrow_S \pi_1$. Finally, as $\pi_2 \rightarrow_{\mathcal{H}} \pi_3$, we should have $\pi_2 \rightarrow_S \pi_3$ that leads to a contradiction, meaning that there cannot be any equivalent sequential execution.

3.3 No One-Size-Fits-All TM Algorithm [77]

It is important to notice that the aforementioned expressiveness limitation is not related to a particular consistency criterion but to the transaction abstraction itself. We used a common consistency criterion, called opacity, offered by many transactional memory algorithms, to show this limitation, however, other transactional memories offer different semantics. In this section, we illustrate our results [77] obtained while testing the semantics of existing STM libraries with our unit-testing framework, called TMunit [75].

3.3.1 Some transaction semantics

Here are examples of correctness properties that are often targeted by transactional memories:

- Serializability is a consistency criterion [123] that has been extensively used to characterize transactional database systems. Serializability requires that any execution of the system, usually involving transactions with reads and writes, must appear equivalent to an execution where all its transactions (i.e., corresponding sequences of reads and writes) would have executed sequentially.
- Linearizability was defined for non-transactional operations of various types [91]. The application of linearizability to transactional operations, formally defined earlier (§§§3.1.2), requires that all executions restricted to committed transactions must be equivalent to an execution where transactions would be executed sequentially and where non-concurrent transactions must be in the same (real-time) order.
- Opacity aims at preventing transactions from accessing inconsistent states that would result in division-by-zero errors or infinite loops [72]. Opacity requires that all committed and aborted transactions appear as if they were executed atomically in an order satisfying the real-time

order (like linearizability applied to reads and writes), this ordering requirement is often called *strictness* [18]. Opacity also requires that all transactions including aborted ones access only consistent system states at any time.

- Single global-lock atomicity (SGLA) is a criterion simple to reason with as it describes the semantics of transactions to be as if every transaction acquires a single global lock for its whole execution [113].
- Virtual world consistency requires that all committed transactions appear as if they were executed atomically in an order satisfying the order of non-concurrent transactions and that all transactions including aborted ones access only consistent system states at any time. Hence, this criterion is weaker than opacity as it allows the set of all transactions (aborting and committing) to not appear as executed atomically, as long as the subset of committed transactions does so [94].

We tested a series of 6 distinct transactional memories against semantic tests that outlined discrepancies among the transactional memories [77]. Interestingly, we noticed that transactional memory algorithms may be overly conservative due to their inherent design, because they are object-based hence protecting with a granularity defined by the size of the considered objects. Others, like SwissTM [47], would be overly conservative in some situations to obtain higher performance in other situations by having a lock protecting multiple memory locations, hence being a source of false sharing. Some, like TL2 [40], would abort in case of a write access on x concurrent with a read-only transaction that reads the freshly written value of x, even though this may not be necessary. Finally, we confirmed that WSTM [78] violates opacity and virtual-world consistency: Since opacity and virtual-world consistency both require that linearizability of reads and writes be satisfied and that transactions do not interfere with aborting transactions whereas WSTM is known to let a write interfere with an ongoing reading transaction.

3.3.2 The inherent limitation of transactions

Even though existing transactional algorithms offer different semantics, it is noteworthy that our expressiveness limitation is not tied to a particular consistency criterion but actually stems from the transaction syntax regardless of its semantics. Consider a snapshot type [6] whose transitions $\{q, \pi, q', r\} \in \Delta$ are depicted in Table 6 rather than a set type in the execution example of Table 5. A τ'-snapshot type exports an $\texttt{update}(i, e)$ operation that writes value $e \in \tau'$ at the element at index $i \in \mathbb{N}$ and a `scan` operation that collects an instantaneous view of all the values.

π	`scan()`	$\texttt{update}(i, e)$
q	$r = q \wedge q' = q$	$r = \texttt{ok} \wedge q'[i] = e \wedge \forall j \neq i : q'[j] = q[j]$

Table 6: The sequential specification of the τ'-snapshot type with q the current state, $\pi = \{\texttt{scan}(), \texttt{update}(i, e)\}$, $i \in \mathbb{N}$, $e \in \tau'$, r the returned value and q' the resulting state

Replace accordingly in Table 5 the `contains` operation by a `scan` operation and the two `insert` operations by two `update` operations. (Note that one can use a linked list to implement a snapshot type simply by considering that each node maintains an index i and an associated value e, the `update` may add a new node or replace the value of an existing node and the `scan` returns the values of all the indices present in the list as we will describe later in the right-hand side of Algorithm 2.) This resulting new execution α' is not correct: the $\pi_1 = \texttt{scan}$ observes the $\pi_3 = \texttt{update}$ of p_3 but not the $\pi_2 = \texttt{update}$ of p_2 whereas the latter completes before the former starts. In other words, we have a history $\mathcal{H}$ where $\pi_2 \rightarrow_{\mathcal{H}} \pi_3$. Hence the condition (1) of linearizability (§§§3.1.2) should ensure instead that the changes of p_2 be visible from the `scan` operation or, said differently, that $\pi_2 \rightarrow_S \pi_1$ in a supposedly equivalent

sequential history S. This leads to a contradiction as the same sequential history should have $\pi_3 \rightarrow_S \pi_1$, hence H violates linearizability.

To conclude, the crux of the problem stems from the fact that classic transactions are oblivious to the operation they protect: they have to adopt an overlying conservative technique to protect an operation by enclosing it while ignoring the semantics corresponding to this operation. Example of how transactions can be split within an operation will be given later (§§4.3).

3.4 Conclusion

The aforementioned concurrency limitation is inherent to the classic transaction model. In particular, this problem cannot be addressed simply by relaxing the consistency criterion a particular transactional memory ensures because this same transaction memory would not be usable to implement any method that requires a stronger consistency criterion. As we explore below, an alternative transaction model with a combination of transactions of different forms can be designed to bypass this limitation while retaining the simplicity of transactions (§§4.1, §§4.2). Although this alternative model seems necessary to achieve good performance, we will see that this may not be sufficient and it might be necessary sometimes to rethink the application itself (§§4.3).

4. Enhancing Concurrency

One can view the concurrency limitation of transactions as the price of bringing concurrency to the masses and making it possible for average programmers to write parallel programs that use shared data. However, some programmers are concurrency experts and these might find it frustrating not to be able to use their skills for enhancing performance. We show that this price can be avoided by combining multiple transaction forms of distinct semantics.

4.1 Relaxing the Transactional Model [54]

As we described with the example of transactionalizing a `contains` and a `scan`, the crux of the limitation of transaction is that they cannot exploit the semantics of the application. In this section, we present our *elastic transaction* model that we proposed to take the semantics of some applications, like search structures, into account. While not presented here, we obtained better performance than with the classic model in different programming languages (C/C++ and Java) and on various architectures (our manycore SCC [64], on our UltraSPARC T2 [61] and on our 16-core AMD machine [54]). Upon conflict detection, an elastic transaction might drop what it did so far within a separate transaction that immediately commits, and resumes its computation within a new transaction, which might itself be elastic. Just like for a regular transaction, the programmer must simply delimit the blocks of code that represent elastic transactions.

4.1.1 The elastic transaction model

Differently from most transaction models, during its execution, an elastic transaction can be *cut* (by the elastic transactional memory) into multiple regular transactions, depending on the conflict it encountered at runtime. More specifically, upon conflict detection an elastic transaction decides whether it can cut itself; if so it commits the past accesses as if they were part of a regular transaction before resuming into a continuation transaction until it encounters a new conflict or commits. A cut is prohibited if during the interval where the elastic transaction executes a pair of consecutive accesses on two locations, these two locations get individually updated. Only in this rare case does the elastic transaction abort. In other cases, a cut could cause the elastic transaction to execute a constant number of additional accesses before committing the past ones. In a sense, these few extra accesses can be viewed as a partial roll-back that is the price to pay to avoid aborting the elastic transaction.

Consider the previous example (Table 5). The programmer, say Alice, has simply to label the transaction of the `contains` with the keyword "elastic" so that the schedule of Table 5 gets accepted by the list-based set implementation. Then, the aforementioned history $\mathcal{H}$ (§§§3.2.2) can be viewed as another history $\mathcal{H}'$ that results from the combination of several pieces:

$$\mathcal{H}' = \boxed{r(h)^1, r(x)^1}^{s_1}, r(h)^2, r(x)^2, w(h)^2, r(h)^3, r(n)^3, r(x)^3, r(z)^3, r(t)^3, w(z)^3, \boxed{r(z)^1, r(m)^1}^{s_2}.$$

In $\mathcal{H}'$, the elastic transaction of the `contains` operation has been cut into two transactions s_1 and s_2. Crucial to the correctness of this cut, no two modifications on x and z have occurred between $r(x)^{s_1}$ and $r(z)^{s_2}$. Otherwise the transaction would have to abort. Now that Alice has the choice between using an elastic transaction or a classic transaction, she can also improve the concurrency of her program when compared to a classic transactional program. In fact, it is easy to see that the schedules rejected by elastic transactions would also be rejected if, in the same implementation, these transactions were opaque (§§§3.3.1), hence elastic transactions allow Alice to derive an implementation that allows for strictly more concurrency than with classic transactions.

Algorithm 2 Elastic transaction `contains` and classic transaction `scan`

```
1:  tx-contains(val)_p:
2:      int result;
3:      node *prev, *next;
4:      curr = set→head;
5:      transaction(elastic) {
6:          next = curr→next;
7:          while next→val < val do
8:              curr = next;
9:              next = curr→next;
10:         result = (next→val == val);
11:     }
12:     return result;
```

```
13: tx-scan(collection)_p:
14:     int result;
15:     node *prev, *next;
16:     curr = set→head;
17:     transaction(normal) {
18:         next = curr→next;
19:         while next→val ≠ set→tail→val do
20:             curr = next;
21:             next = curr→next;
22:             result = result ∪ {curr→val}
23:     }
24:     return result;
```

Elastic transactions cannot, however, be used in all cases and it is the task of Alice to decide when she can use elastic transactions for the program to remain correct. For example, consider the code of a `scan` operation on a snapshot object. Recall that as part of the snapshot type, this `scan` operation must return an instantaneous snapshot of all elements present as depicted in Table 6. The elastic transaction cannot be used to protect the `scan` operation as it could violate linearizability of the snapshot object. To illustrate this violation, consider again the example of Table 5 where the `contains` is replaced by a `scan` labelled as an elastic transaction and the two `inserts` are replaced with `updates` labelled as classic (opaque) transactions.

As the elastic transaction is oblivious to the operation semantics, the history will, as in history $\mathcal{H}'$, be as if the elastic transaction cut itself into two sub-transactions s_1 and s_2. This cut would, however, allow the $\pi_1 =$ `scan` operation to observe the $\pi_3 =$ `update` by process p_3, but not the $\pi_2 =$ `update` by process p_2, hence implying that an equivalent sequential history S should have $\pi_3 \rightarrow_{\mathcal{S}} \pi_1$ and $\pi_2 \nrightarrow_{\mathcal{S}} \pi_1$, however, $\pi_2 \rightarrow_{\mathcal{H}'} \pi_3$ would violate condition (1) of linearizability. In other words, the scan would return an inconsistent snapshot including a recently updated element but not the older updated element, hence violating the sequential specification of the snapshot type. If, instead, Alice delimits the `scan` operation as a "normal" transaction, as depicted in the right-hand side of Figure 2, then the `scan` would execute as a classic transaction that is opaque and the incorrect schedule would be rejected, because one of the three transactions would abort. Note that a novice programmer, say Bob, can always safely use classic transactions in place of elastic transactions, if unsure.

It is interesting to note that the cuts of elastic transactions enable more concurrency than what an expert programmer could do with classic transactions. First, the cuts are dynamically tried at runtime depending on the interleaving of accesses. As this interleaving is generally non-deterministic, a programmer cannot just split transactions prior to execution and ensure correct executions.

4.1.2 Advantages of elastic transactions

Other transactional models also allow the programmer to reason in terms of the implemented type to ignore read-write conflicts as long as there are no conflicts between operations. Commutativity [100, 130, 144], a binary relation over operations, was extensively used to increase inter-transaction concurrency.

The consistency criterion called multi-level serializability [145] applies to operations at different levels of abstraction, thus expressing the impact low level `reads` and `writes` may have on the higher-level operations that comprise them. More precisely, ℓ-level serializability is defined recursively assuming that any history is 0-level serializable. A history that is ℓ-level serializable and in which we can find a

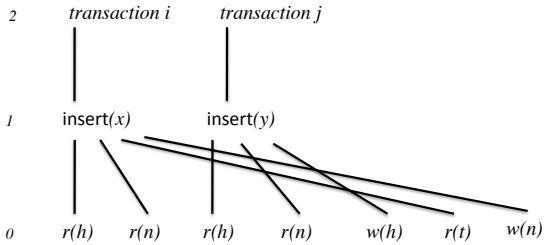


Figure 3: A history where operations are represented at different levels of semantics.

serialization of level $\ell + 1$ operations by commuting level ℓ operations, is also $(\ell + 1)$-level serializable. Identifying two operations that commute at level ℓ is thus used to define an $\ell + 1$ level serialization.

Consider, using Weikum's hierarchy [145], a list-based set execution involving two `inserts` as depicted in Figure 3. The two transactions i and j appear at level 2 of this hierarchy, the integer set operations called by transactions i and j appear at level 1 and the read and write operations they called appear at level 0. The history is neither 2-level nor 1-level serializable because of two dependencies: between $r(h)^i$ and $w(h)^j$ and between $r(n)^j$ and $w(n)^i$, however, both would safely commit if i was elastic. In fact, elastic transactions commute operations that would not commute with multi-level serializability by exploiting the interleaving information obtained at *runtime*.

Alternatively, the programmer can exploit commutativity on search structures using transaction models like open nesting [118, 119, 140] and transactional boosting [86] or parallelization models dedicated to irregular applications, like the Galois system [101]. These models are flexible as they let the programmer define additional abstractions other than set or dictionary types, however, they all require the programmer to specify explicitly the commutativity of operations and abort handlers with appropriate compensate actions. These abort handlers could be as long as the transactions themselves, hence, making programming with these transaction models potentially error prone.

Elastic-opacity. As soon as a transactional memory system offers elastic transaction, it can no longer guarantee opacity. We defined a new consistency criterion that is satisfied by elastic transactions called *elastic opacity*. It applies to high level operations that require weaker guarantees than the previous proposed criteria. In short, elastic-opacity assumes a model with two types of transactions, elastic and normal, and requires that if we cut elastic transactions into smaller sub-transactions, then we obtain an execution composed of normal transactions and sub-transactions that is opaque. Elastic-opacity guarantees operation atomicity at the application level, its goal is to enhance concurrency by relaxing the unnecessary atomicity provided at the read/write level. Hence, elastic-opacity is a strictly weaker consistency criterion than opacity.

There are various ways of implementing elastic opacity (and an elastic transactional memory). One can exploit existing functions that extend the interface of the transactional memory system: `light reads` [7], `unit-load` [53] and early `release` [88]. Early `release` can be used explicitly by the programmer to indicate from which point of the transaction all conflicts involving its read of a given location can be ignored [88]. View transactions [7] propose `light-read` as a language construct that provides a lightweight version of read operation similar to a `unit-read`. It comes in complement to view-pointers used to keep track of read locations forming a critical view the transaction has to revalidate. We implemented the elastic transaction model with `unit-loads` and observed better performance than with our own implementation of elastic transaction, $\mathcal{E}$-STM. The drawback is that we had to change the sequential code of the application.

We implemented $\mathcal{E}$-STM, an elastic transactional memory library in C/C++ and Java and ported it on our manycore SCC [64], on our UltraSPARC T2 [60] and on our 16-core AMD Opteron [54]. We observed higher performance than classic transactional memory libraries, like TinySTM [53] and LSA [125], implemented in the corresponding language [54,64,77]. Elastic opacity was shown sufficient to implement a linearizable list-based set [54] and we believe that the same proof could be generalized to other search structures. The elastic transaction model improves concurrency for a certain kind of semantics but cannot be used in all applications. One can expect that other models can fit other desirable semantics, hence allowing to achieve even higher concurrency and possibly higher performance. We actually present below a single polymorphic transactional memory system that offers three transaction models suited for different semantics. For these different transactions to access the same shared data, it is however crucial to make sure that transactions do not violate the semantics of one another.

4.2 Combining Transactional Models [61]

To illustrate the feasibility of combining multiple transaction semantics in the same transactional memory system, we implemented a polymorphic STM, namely *PSTM*, in Java. PSTM offers transactions with different semantics to the programmer while guaranteeing that all transactions can access concurrently the same shared data without violating the semantics of each other.

Intuitively, the more semantics a polymorphic transactional memory system provides, the more control it gives to experts, allowing them to potentially boost performance even more. The opacity semantics of classic transactions benefits the novice programmer as it is always safe to use. The elastic transactions can bring added performance in search structures (§§4.1). Thus, it seems interesting to complement these two semantics with additional ones. Typically, a `scan` operation of a snapshot object, as we previously described on the right-hand side of Algorithm 2, could benefit from a semantics that allows concurrent updates to overwrite the elements it reads.

4.2.1 Snapshot transaction

In the database context, a transaction can ignore some conflicts yet ensure *snapshot isolation* as this property allows a transaction to commit provided that the values it has written have not been overwritten [52]. In some snapshot isolated databases, the programmer can statically `Select For Update` to avoid the *write-skew* problem [17] where two transactions read memory locations before they update distinct locations among the previously read ones: both transactions read the same value while one should not. Such a construct makes the reads visible without distinction at runtime. In the context of transactional memory, the programmer can statically call specific actions within a transaction to differentiate protected read calls [31].

Instead, we defined a *snapshot* transaction as a transaction semantics or *form* that allows read-only operations to run concurrently with updates. Similarly, to elastic and classic transactions, using a snapshot transaction simply requires to delimit a region of code (and instrument in the same way all the accesses within this region). Unlike explicit memory access calls (e.g., early release), defining a new transaction allows us to require that snapshot transactions be read-only. In particular, snapshot transactions are immune to the write-skew problem because after reading concurrently they can never update disjoint subsets of these data.

A snapshot transaction is particularly appealing for protecting an operation that returns a result that depends on numerous elements of a data type, like a Java Iterator. As an example, a snapshot transaction implementing a size operation is depicted in Algorithm 3.

At first glance, providing as many forms as possible in a single toolbox system may seem the key solution to help develop concurrent applications once and for all, the challenge lies however in the mixture of these semantics. Mixing these semantics requires to let them access the same shared data

concurrently. In fact, it is crucial that the semantics of each individual transaction is not violated by the execution of concurrent transactions of potentially different semantics.

Algorithm 3 Java pseudocode of the size() operation with snapshot transactions

```
1:  public int size():
2:      transaction(snapshot) {
3:          int n = 0
4:          Node⟨E⟩ curr = head
5:
6:          while curr ≠ null do
7:              curr = curr.getNext()
8:              n++
9:      return n
10:     }
```

Combination of snapshot and elastic transactions. To obtain a highly concurrent snapshot semantics, we exploit multi-version concurrency control that lets snapshot transactions commit while concurrent (elastic or classic) updates commit. The implementation of this snapshot transaction is to exploit a global counter and a version number per written value, so that the transaction can fetch the counter at start time and decide while reading new locations to return a value that has an appropriate (not too recent) version consistent with this start time. The combination of the snapshot with classic and elastic transactions in the same application requires, however, to make sure all updates (elastic and classic) record the current value (as a backup) before overriding it.

The combination problem might even be subtler if a relaxed transaction ignores a conflict that involves a concurrent stronger transaction that cannot ignore it. Typically, elastic and opaque transactions handle this issue for read-write conflicts by requiring that only the reading transaction decides upon the conflict resolution. Unlike writes, reads are idempotent so that the semantics of the writing transaction is never altered by the outcome of the conflict resolution. Our solution relies on (1) having invisible reads so that the writing transaction does not observe the conflict and (2) enforcing commit-time validation so that the reading transaction always detects the conflict. Note that it would not be possible for a relaxed transaction to ignore a write-write conflict that cannot be ignored by a strong transaction, so PSTM enforces that one of the two write-write conflicting transactions aborts, however, it does not enforce that the strong transaction always abort in such cases, otherwise it would be unfair.

4.2.2 Reusability

Abstract data types (ADTs) promote (a) *extensibility* when an ADT is specialized through, for example, inheritance by overriding or adding new operations, and (b) *composability* when two ADTs are combined into another ADT whose operations invoke the original ones. Key to this reusability is that there is no need to know the internals of an ADT to reuse it: its interface suffices. With the latest technology development of multicore architectures many programs are expected to scale with a large number of cores: ADTs need thus to be shared by many threads, translating into concurrent data types (CDTs). CDTs are however usually not reusable: the programmer can hardly build upon them. Some of the problems that prevent Bob from reusing the CDTs of Alice's library are called inheritance anomalies [112], however, we showed that it may not be desirable to remedy these particular anomalies [67], thus we concentrate on the more general reusability problem.

The extensibility issue. A first challenge is to guarantee that Bob can extend Alice's type with new operations without having to change Alice's code. In Java, the `ConcurrentLinkedQueue` type of the

JDK 7 exports an inconsistent `size` operation. The problem comes from the fact that this CDT aims at implementing the non-blocking algorithm from Michael and Scott designed to provide efficient `offer` (i.e., push) and `poll` (i.e., pop) [117] but aims also at implementing the `Collection` type including a `size` operation for a neat integration in the Java API. On the one hand, a `size` operation is useful to count the number of elements comprised in this collection: although `size` remains optional, various `Collection` CDTs do provide it. On the other hand, the algorithm of Michael and Scott was optimized to export non-blocking `offer` and `poll` without aiming at supporting a `size` operation or allowing extensibility. This lack of extensibility, which is inherent to the synchronization used, led expert programmers to implement a non-atomic `size` operation. In other words, the resulting CDT does not satisfy the sequential specification of the `Collection` type.

Specifically, this `size` consists of traversing the underlying linked list from the head to the tail while elements are pushed at the head and popped at the tail. Assume that some elements are moved from the tail to the head, one after the other, so that the size s changes by ± 1. As the `size` operation does not protect the head and the tail of the queue, it simply ignores any of these moved elements and can return an incorrect value way smaller than $s - 1$. Precisely because predicting the outcomes of this `size` requires to understand the implementation internals, the resulting CDT is not reusable. We reported this `ConcurrentLinkedQueue` issue to the JSR166 expert group in May 2010. Following up our report, this unexpected behavior has been warned in the documentation of the class `ConcurrentLinkedQueue` on the JSR166 site since revision 1.54 and the issue was still present in the JDK 7. Since then other researchers unaware of this warning observed the same problem [25].

This `size` problem simply illustrates the more general lack of extensibility. One may think of using `ArrayBlockingQueue` to obtain a correct `size` that returns the current value of a counter, however, such a `size` implementation requires to modify all insertion and removal operations to make them adjust the counter. Apart from the `size` example, a programmer would have similar problems as soon as she tries to extend these CDTs with, for example, a `scan` operation to obtain a new kind of queue-snapshot type.

The composition issue. A second challenge is the composition of the semantics. Bob can directly nest Alice's elastic transactions into another transaction thus guaranteeing atomicity and deadlock-freedom of its own operation as depicted in Figure 4. Typically, Bob may use various semantics for his transaction while enclosing Alice's, choosing between labelling his transaction as elastic, snapshot, or classic. For example, one can imagine that Alice provides an elastic `contains`(x) that Bob composes into a snapshot `containsAll`(C) operation that returns successfully only if all elements of a collection C are present. For the sake of safety, we decided that the strongest semantics of the involved transactions, in this case the snapshot one, applies to all. Following the same idea, a novice, unaware of the various semantics, will always obtain a safe composite transactional operation by omitting the semantics that will be classic (opaque) by default. This opaque semantics will automatically be conveyed to inner transactions whatever semantics these inner transactions had initially.

Note that we investigated more formally the composition of relaxed transactions [63] but the composition of transactions of different semantics remains a hard problem and it is unclear whether our safe approach is better suited than an approach favoring concurrency. In particular, deciding upon which semantics to apply for a composite transaction that nests transactions of incomparable semantics remains, as far as we know, an open question.

A typical example of composition bug that affects other synchronisation technique is due to the use of locks for synchronization in Java CDTs is the `java.util.Vector` class of Java 1.4.2. As noted earlier [57, 143], the Vector class, which is a widely used abstraction that is supposed to be thread-safe, suffered from a critical issue related to one of its constructors. Upon constructing a new `Vector` based on an existing `Collection c` of objects, an `ArrayOutOfBoundsException` could be raised. The reason is that between the time the `size` of the collection `c` is computed and the time `c` gets converted into an array, a concurrent update could modify the size of the collection `c`. This problem had been reported and

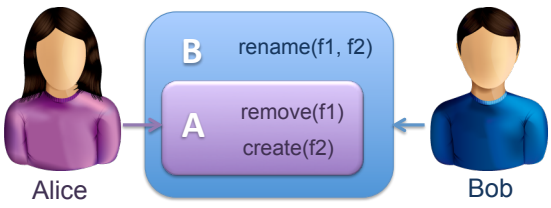


Figure 4: Bob composes Alice's component operations remove and create into a new operation rename

fixed in later versions of the JDK but illustrates the difficulty of composition with other synchronisation techniques.

The polymorphic transaction methodology. Figure 3 illustrates how to exploit our *polymorphic transaction* methodology to cope with the `ConcurrentLinkedQueue` issue and to propose an alternative `ReusableLinkedQueue` easy to reason about. It requires that the operations `pop` and `push` accessing mutable shared variables use no explicit synchronizations besides transactions. In this particular example, the size variable is used to count all elements as a sequential `size` operation delimited with a transaction of the snapshot form.

The resulting implementation is inherently extensible. The snapshot transaction form guarantees that all shared read accesses of the `size` operation, including the one to `getNext()`, return values present at a common point in time between the invocation and the response of `size`. To this end, the implementation associates a version to each value written by any transaction, a snapshot transaction records the highest version at start time and identifies the correct value to return upon reading based on the associated version. In particular, all updates to mutable shared variables are tracked using metadata so that `size` can detect that a field of `ReusableLinkedQueue` is being or has been overridden by a concurrent operation (e.g., `offer` or `poll`) and choose to return an older version of the field or to abort (if no suitable version exists). Note that one could have safely omitted the snapshot form parameter here (`@Transactional`) hence adopting the default opaque semantics instead, however, if such a semantics does not use multiversion concurrency control then it would limit concurrency by often aborting the `size` or its potential conflicting updates.

The aforementioned `java.util.Vector` composition issue can be easily fixed using our polymorphic transaction methodology that instruments all transactional shared accesses (including to the `Collection`). The obtained `ReusableOldVector` simply consists of the original constructor placed into the `init` operation that is annotated with a keyword `@Transactional(form=OPAQUE)` [61]. To evaluate the performance of our methodology, we also implemented a `ReusableVector` CDT by converting all the `synchronized` operations of the `java.util.Vector` of the JDK 7 (hence the name `ReusableOldVector` for the fix of the version 1.4.2) into sequential operations annotated using the `opaque` transactional wrapper. Under contention, the `ReusableVector` is slower than the original `Vector`, in turn, slower than the non-synchronized sequential version of the Vector (running sequentially). Under read-only workloads, `ReusableVector` outperforms `Vector`. Another advantage of our transactions is that each operation, be it private (e.g., `ensureCapacityHelper`) or public (e.g., `ensureCapacity`) can be annotated as a transaction. In contrast, nesting locks may lead to deadlocks when a programmer encapsulates in a `synchronized` block a call to an external operation already using `synchronized`.

Deadlock freedom. A large body of work aimed at simplifying concurrent object-oriented programming. However, most of it inherits the problem of lock-based programs as they rely on locks or monitors like Guava [13]. In particular, SCOOP allows to specify an object accessed by a different process as

separate [115]. SCOOP was ported to Java [139] but is not inherently deadlock-free [147].

The first TM to handle concurrency in a dynamic control flow redirects speculative accesses to Java object copies [87, 88]. Lightweight transactions were suggested to avoid copying entire objects by using a mapping of addresses to word-sized ownership records [78] before field-based instrumentation was proposed [99]. Two interesting ways to bypass the concurrency limitations (§§3.2) of TMs are to use abstract locks—in open nesting [118] and transactional boosting [86]—that protect the abstraction during the execution of a type operation. The issue of such solutions is that upon abort, another lock acquirement may be needed by a compensating operation to undo the effects of the operation, which may lead again to deadlocks [119].

The way we avoid these deadlock situations in the polymorphic transaction methodology is by adopting an optimistic strategy where a transaction acquires the locks only when it is guaranteed to commit.

Results. Using our polymorphic transaction methodology, we implemented three reusable types (i.e., queue, set and map) implemented with four data structures (i.e., linked list, skip list, hash table and array), synchronized with PSTM and tested them against lock-based and non-blocking alternative implementations from the Java Development Kit (including the `java.util.concurrent` library) and classic transaction-based implementations on our UltraSPARC T2 and our 64-core AMD Opteron machines (cf. Table 1). On some workloads, our implementation speeds up the alternative solution that relies on the Java copyOnWrite for synchronization by 2.4× on average, and the existing reusable pessimistic lock-based solution by 4.7× on average at the highest level of parallelism we had at our disposal (64 hardware threads). Moreover, our solution outperforms techniques relying on common monomorphic STMs by up to a factor of 8.6× on 64 threads but we observed that non-blocking but non reusable techniques from the JDK (especially the implementation in Java of the Michael and Scott queue [117]) could be 3× faster than our PSTM-based queue. We consider this difference in performance to be part of the price to pay for reusability.

Polymorphic transactional memory allows an expert programmer to tune the performance of her library while the novice programmer can reuse it almost as simply as a sequential library because the library satisfies the sequential specification of the targeted type. To efficiently reuse a transactional library, however, it is sometimes necessary to be a knowledgable programmer to understand the library implementation before properly adapting the use of transactions to the library to maximize its concurrency, as we explain below.

4.3 Rethinking Applications [33]

In this section, we propose to rethink the application that is synchronized with transactions to maximize concurrency. Given what we have proposed so far, one may be tempted to think that writing a highly concurrent transactional data structure simply requires to design a sequential data structure with polymorphic transactional delimiters. As we explain now, however, one can gain in concurrency even further by designing a data structure keeping in mind that transactions should be as short as possible to favor concurrency. Below we present the speculation-friendly tree we designed to minimize transaction size [33].

4.3.1 Optimism vs. pessimism

The programmer should keep in mind whether the implementation she is targeting is optimistic or pessimistic. A *pessimistic* implementation captures what can be achieved using classic conservative locks

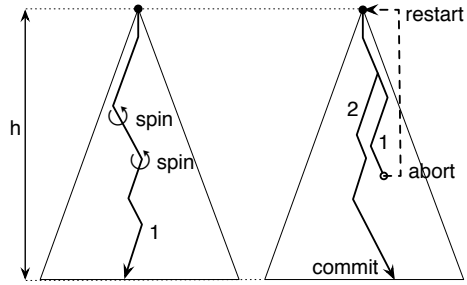


Figure 5: A balanced search tree whose complexity, in terms of the amount of accessed elements, is **(left)** proportional to h in a pessimistic execution and **(right)** proportional to the number of restarts in an optimistic execution

like mutexes, spinlocks, reader-writer locks. In contrast, an *optimistic* implementation proceeds speculatively and may roll back in the case of conflicts, e.g., relying on classical TMs, like TinySTM [53], or more relaxed forms of optimistic techniques, such as "lazy" synchronization [83] and elastic transactions [54].

To illustrate the difference between optimistic and pessimistic implementations, consider the example of Figure 5 depicting their step complexity when traversing a tree of height h from its root to a leaf node. On the left, steps are executed pessimistically, potentially spinning before being able to acquire a lock, on the path converging towards the leaf node. On the right, steps are executed optimistically and some of them may abort and restart, depending on concurrent thread steps. The pessimistic execution of each thread is guaranteed to execute $O(h)$ steps (possibly spinning at times, waiting on locks), yet the optimistic one may need to execute $\Omega(hr)$ steps, where r is the number of restarts. Note that r depends on the probability of conflicts with concurrent transactions that depends, in turn, on the transaction length and h. When implementing a tree using TM, although it is clear that a transaction must be aborted before violating the type (e.g., set) implemented by this tree (e.g., inserting k successfully in a set while k was already present), it is unclear whether a transaction must be aborted if there is a risk that it unbalances slightly the tree (e.g., making the largest paths from the root to a leaf strictly more than twice larger than the shortest path from the root to a leaf).

Large vs. small transactions. In this section, we focus our attention on the structural invariant of existing binary tree libraries, namely the *balance*, and enlighten the impact of their restructuring, namely the *rebalancing*, on contention.

Binary trees provide logarithmic access time complexity given that they are balanced, meaning that among all downward paths from the root to a leaf, the length of the shortest path is not far apart the length of the longest path. Upon tree update, if their difference exceeds a given threshold, the structural invariant is broken and a rebalancing is triggered to restructure accordingly. This threshold depends on the considered algorithm: AVL trees [3] do not tolerate the longest length to exceed the shortest by 2 whereas red-black trees [15] tolerate the longest to be twice the shortest, thus restructuring less frequently. Yet in both cases the restructuring is triggered immediately when the threshold is reached to hide the imbalance from further operations.

Generally when writing an algorithm using transactions, one takes an existing tree algorithm and encloses each of its operations within an opaque transaction to obtain a concurrent tree whose operations are guaranteed atomic (i.e., linearizable), however, the obtained concurrent transactions likely *conflict* (i.e., one accesses the same location another is modifying), resulting in the need to abort one of these transactions, leading to a significant waste of efforts. This is in part due to the fact that enclosing an

successful update operation (i.e., a successful `insert` or a `remove` operation) into a transaction boils down to enclosing four phases in the same transaction:

1. the modification of the type state,
2. the corresponding structural adaptation,
3. a check to detect whether the invariant risks to be violated and
4. the potential rebalancing if needed.

Note that it is easy to find a schedule rejected by a tree implementation synchronized with classic (opaque) transactional memory, even though this schedule could seemingly have been exported by a correct execution. Consider an `insert(k)` and a `delete(k)` concurrently executed by two processes. Assume that `insert(k)` executes its steps (1)-(3) and has to rebalance but then gets delayed before it can execute step (4). If the `delete(k)` executes its steps (1) and (2), then the step (4) tentatively updating node with key *k* again, will create a cycle in the conflict graph of low level reads and writes. Although this would not violate the linearizability of the set type implemented by this tree, the aforementioned schedule will be typically rejected by classic transactional memory implementations for the same reason as why the previous schedule involving the `contains` operation was rejected (§§3.2).

A transaction-based red-black tree. An example of a tree implementation rejecting these correct schedules is the transaction-based binary search tree developed by Sun Microsystems and other researchers. It was extensively used to evaluate transactional memories [26, 40, 46, 53, 86, 88, 149]. This library relies on the classical red-black tree algorithm that bounds the step complexity of pessimistic `insert/delete/contains` of the set type (cf. Table 4). It has been slightly optimized for transactions by removing sentinel nodes to reduce false-conflicts, and we are aware of two benchmark-suite distributions that integrate it, STAMP [26] and Synchrobench [59].

Each of its update transactions encloses all four phases given above even though phase (1) could be decoupled from phases (3) and (4) if transient violations of the balance invariant were tolerated. Such a decoupling is appealing given that phase (4) is subject to conflicts. The reason for this is that in phase (4), the algorithm balances the tree by executing rotations starting from the position where a node is inserted or deleted and possibly going all the way up to the root. A rotation consists of replacing the parent node where the rotation occurs by one of its children and adding this parent node to one of its subtrees. Not only do these modifications create conflicts with other transactions but a node cannot be safely accessed concurrently by an abstract transaction (or an operation) and a rotation, otherwise the abstract transaction might miss the node it targets while being rotated downward. Similarly, rotations cannot access common nodes as one rotation may unbalance the others.

Moreover, the red-black tree does not allow any abstract transaction to access a node that is concurrently being deleted from the abstraction because phases (1) and (2) are tightly coupled within the same transaction. If this was allowed the abstract transaction could end up on the node that is no longer part of the tree. Fortunately, if the modification operation is a deletion then phase (1) can be decoupled from the structural modification of phase (2) by simply marking the targeted node as logically deleted in phase (1), effectively removing it from the set abstraction prior to unlinking it physically in phase (2). This improvement is important as it lets a concurrent abstract transaction travel through the node concurrently being logically deleted in phase (1) without conflicting. Making things worse due to the "all or nothing commit/abort" semantics of transactions, without decoupling these four phases, having to abort within phase (4) would typically require the three previous phases to restart as well. Finally, without decoupling, only `contains` operations are guaranteed to not conflict with each other. With decoupling, `insert/delete/contains` do not conflict with each other unless they terminate on the same node.

To conclude, for the transactions to preserve the linearizability of the type and invariants of such a tree algorithm, they typically have to keep track of a large *read set* and *write set*, i.e., the sets of accessed memory locations that are protected by a transaction. Possessing large read/write sets increases the probability of conflicts and thus reduces concurrency. This is especially problematic in trees because the distribution of nodes in the read/write set is skewed so that the probability of the node being in the set is much higher for nodes near the root (e.g., the root is guaranteed to be in the read set).

4.3.2 A speculation-friendly tree

We proposed a *speculation-friendly* binary search tree that implements a dictionary type, with multiple key-value pairs. As depicted in Table 7, the dictionary type with transitions $\{q, \pi, q', r\}$ is similar to the set type depicted previously in Table 4, except that each key is unique whereas each element is a key-value pair. In short, the novelty of this tree is to reduce the size of transactions compared to the red-black tree from Sun Microsystems by decoupling the tree rotation.

π	$\texttt{update}(k, v)$	$\texttt{delete}(k)$	$\texttt{get}(k)$
$\langle k, * \rangle \in q$	$r = \texttt{true} \wedge q' = (q \setminus \{\langle k, * \rangle\}) \cup \{\langle k, v \rangle\}$	$r = \texttt{true} \wedge q' = q \setminus \{\langle k, * \rangle\}$	$r = v \wedge q' = q$
$\langle k, * \rangle \notin q$	$r = \texttt{true} \wedge q' = q \cup \{\langle k, v \rangle\}$	$r = \texttt{false} \wedge q' = q$	$r = \bot \wedge q' = q$

Table 7: The sequential specification of the τ'-dictionary type with q the current state, $\pi = \{\texttt{update}(k, v), \texttt{delete}(k, v), \texttt{get}(k)\}$, $k \in \tau'$, r the returned value and q' the resulting state

The motivation for rotation decoupling stems from two separate observations: (i) a rotation is tied to the modification that triggers it, hence the process modifying the tree is also responsible for ensuring that its modification does not break the balance invariant and (ii) a rotation affects different parts of the tree, hence an isolated conflict can abort the rotation performed at multiple nodes. In response to these two issues we introduce a dedicated rotator thread responsible for performing structural adaptations, allowing abstract transactions to complete faster and we distribute the rotations into multiple (node-)local transactions. Note that our rotating thread is similar to the collector thread proposed by Dijkstra et al. [44] to garbage collect stale nodes.

In our speculation-friendly tree, deciding when to perform a rotation is done based on local balance information. This local rotation was introduced in [21] and works as follows: *left-h* (resp. *right-h*) is a node-local variable to keep track of the estimated height of the left (resp. right) subtree. *local-h* (also a node-local variable) is always 1 larger than the maximum value of *left-h* and *right-h*. If the difference between *left-h* and *right-h* is greater than 1 then a rotation is triggered. After the rotation these values are updated accordingly based on the type of rotation performed. Since these values are local to the node the estimated heights of the subtrees might not always be accurate. The propagate operation (described in the next paragraph) is used to update the estimated heights. Using the propagate operation and local rotations, the tree is guaranteed to be eventually balanced [21, 24].

The rotating thread executes continuously a depth-first traversal to propagate the balance information. Although it might propagate outdated height information due to concurrency, in the absence of concurrent modifications, the tree becomes eventually balanced. The only requirement to ensure balance when using propagations and local rotations is that a node knows when it has an empty subtree (i.e., when *node*.ℓ is $\bot$, *node.left-h* must be 0). This requirement is guaranteed by the fact that a new node is always added to the tree with *left-h* and *right-h* set to 0 and that these values are updated when a node is removed or a rotation takes place. Each propagate operation is performed as a sequence of distributed transactions each acting on a single node. Such a transaction first travels to the left and right child nodes, checking their *local-h* values and using these values to update *left-h*, *right-h*, and *local-h* of the parent

node. As a single rotator thread is used and no type operations access these three values, they never conflict with the propagation, meaning that no synchronization is necessary.

Empirical measures of step complexity. To get a rough idea of the effect of decoupling update operations on the step complexity of classical transactional balanced trees we have counted the maximal number of reads necessary to complete typical `insert/delete/contains` or `update/delete/get` operations. Note that this number includes the reads executed by the transaction each time it aborts in addition to the read set size of the transaction obtained at commit time.

Update	0%	10%	20%	30%	40%	50%
AVL tree	29	415	711	1008	1981	2081
Sun red-black tree	31	573	965	1108	1484	1545
Speculation-friendly tree	29	75	123	120	144	180

Table 8: Maximum number of transactional reads per operation on three 2^{12}-sized balanced search trees as the update ratio increases

We evaluated the aforementioned red-black tree, an AVL tree, and our speculation-friendly tree on our 48-core AMD Opteron machine listed in Table 1 using the same TM algorithm, TinySTM-CTL, i.e., with lazy acquirement [53]. We used Synchrobench [59] workloads with from 0% to 50% update while keeping the expectation of the tree sizes fixed to 2^{12} during the experiments by performing an insert and a remove with the same probability. Table 8 depicts the maximum number of transactional reads per operation observed among 48 concurrent threads as we increase the update ratio, i.e., the proportion of insert/remove operations over contains operations.

For all three trees, the transactional read complexity of an operation increases with the update ratio due to the additional aborted efforts induced by the contention. Although the red-black and the AVL trees objective is to keep the complexity of pessimistic accesses $O(\log_2 n)$ (proportional to 12 in this case), where n is the tree size, the read complexity of optimistic accesses grows significantly as the contention increases (14× more at 10% update than at 0%, where there are no aborts).

Our resulting speculation-friendly tree outperforms previous transaction-based AVL and red-black trees by being transiently unbalanced to reduce contention during contention peaks, yet we showed, in an extended version of our original work [33], that $O(n^2)$ elementary steps rebalance the tree when contention stops. In particular, the speculation-friendly tree is shown correct, reusable and speeds up a transaction-based travel reservation application by up to 3.5× on our 48-core AMD Opteron (cf. Table 1) running Linux 2.6.32. Besides for binary search trees, reducing transactions of skip list was also found particularly efficient [48].

4.4 Conclusion

While transactions can be made more concurrent to respond to the needs of performance required by experts and simplicity required by novices, using transactions still has some cost. First, a well-engineered library that is not reusable may offer better performance than reusable alternatives. Second, enhancing concurrency of transactions require to design a polymorphic transactional memory raising questions on the semantic resulting from nested transactions of different semantics—while we only scratched the surface of this problem, this semantics still has to be specified carefully. Finally, the performance of reusable libraries can further be improved by making sure that the application is tuned to allow for reducing conflicts among transactions, hence, rethinking the sequential code is another important ingredient to maximize performance of transactional applications.

5. Conclusion and Future Work

The research we conducted contributed to facilitate the adoption of transactional memory. First, our benchmarking of transactional memory against sequential code allowed us to conclude that software support for transactional memory was a viable solution to compensate hardware limitations while exploiting modern chip multiprocessors, including multicore and manycore platforms. Second, we identified concurrency limitations inherent to the transaction abstraction essentially due to its obliviousness to the application semantics. Finally, we proposed solutions to overcome these issues by proposing a new abstraction, polymorphic transaction, that helps programmers boost the performance of their applications. Transactional memory can now be used by differently skilled programmers to write efficient and simple concurrent programs.

As future work, we envision the study of synchronization, data structures and contention. First, with the introduction of new synchronization techniques comes the question of the choice of the ideal synchronization technique. While we have started comparing experimentally software transactional memory libraries against other synchronization techniques [59] there is a long way before one can identify the best synchronization technique depending on the targeted performance and simplicity. A possible direction is to draw a profile of applications based on selected characteristics to automate the choice of synchronization depending on performance observed in the past with similar applications [62].

Second, our relaxation of the balanced tree invariant (§§4.3) could be generalized to other data structures to lower contention as our preliminary results on skip lists indicated [35, 43]. Most of the sequential data structure algorithms have structural invariants that can be relaxed without affecting correctness. By maintaining this correctness, the simplicity to reason about the implemented specification is preserved. By relaxing the structural invariants one can adjust the asymptotic complexity of the structure to reduce contention and gain in performance.

Third, we reasoned formally about concurrency to find new ways of deriving better performance, however, we omitted important metrics, like contention that becomes a major cause of performance drop as new machines keep offering a growing amount of processing cores. Several notions of contention have been recently defined, the most prominent ones being probably the point contention [11] and the interval contention [8], however, they may be too rough to precisely analyze the amortized complexity one can expect from a concurrent data structure [58].

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP*, 2009.

[2] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *ISCA*, pages 451–461, 2009.

[3] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proc. of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962.

[4] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. E. Saha. Unlocking concurrency: Multicore programming with transactional memory. *ACM Queue*, 4(10), Dec 2006.

[5] 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 *PLDI*, 2006.

[6] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sept. 1993.

[7] Y. Afek, A. Morrison, and M. Tzafrir. Brief announcement: view transactions: transactional model with relaxed consistency checks. In *Proc. of the 29th Annual ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 2010.

[8] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[9] G. Almes, A. Black, E. Lazowska, and J. Noe. The eden system: A technical review. *IEEE Trans. on Software Engineering*, SE-11(1):43–59, 1985.

[10] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *ICA3PP*, 2008.

[11] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM (JACM)*, 50(4):444–468, 2003.

[12] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* John Wiley & Sons, 2004.

[13] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA*, pages 382–400, 2000.

[14] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, pages 29–44, 2009.

[15] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica 1*, 1(4):290–306, 1972.

[16] R. Bayer and M. Schkolnick. *Concurrency of operations on B-trees*, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

[18] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[19] S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.

[20] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[21] L. Bougé, J. Gabarro, X. Messeguer, and N. Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical Report RR1998-18, ENS Lyon, 1998.

[22] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *OSDI*, 2010.

[23] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, 2012.

[24] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP*, pages 257–268, 2010.

[25] J. Burnim, G. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *ASPLOS*, pages 79–90, 2011.

[26] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, 2008.

[27] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58, 2008.

[28] C. Cascaval, C. Blundell, M. M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[29] J. Chen and W. Watson III. Multithreading performance on commodity multi-core processors. http://usqcd.jlab.org/usqcd-docs/qmt/multicoretalk.pdf.

[30] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. *PACT*, pages 155–166, 2011.

[31] C. Cole and M. Herlihy. Snapshots and software transactional memory. *Sci. Comput. Program.*, 58(3):310–324, 2005.

[32] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *Micro, IEEE*, 30(2):16–29, 2010.

[33] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *PPoPP*, pages 161–170, 2012.

[34] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly binary search tree. In *Euro-Par*, pages 229–240, 2013.

[35] T. Crain, V. Gramoli, and M. Raynal. No hot spot non-blocking skip list. In *ICDCS*, pages 196–205, 2013.

[36] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPoPP*, pages 67–78, 2010.

[37] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346, 2006.

[38] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, 2013.

[39] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, pages 157–168, 2009.

[40] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[41] D. Dice and N. Shavit. What really makes transactions faster? In *Transact*, 2006.

[42] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *Transact*, 2009.

[43] I. Dick, A. Fekete, and V. Gramoli. Logarithmic data structures for multicores. Technical Report 697, University of Sydney, 2014.

[44] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[45] M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *POPL*, pages 233–246, 2015.

[46] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011.

[47] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI*, pages 155–165, 2009.

[48] A. Dragojević and T. Harris. STM in the small: Trading generality for performance in software transactional memory. In *EuroSys*, pages 1–14, 2012.

[49] A. Dragojević, Y. Ni, and A.-R. Adl-Tabatabai. Optimizing transactions for captured memory. In *SPAA*, 2009.

[50] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19:624–633, 1976.

[51] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. *Proc. 10th USENIX NSDI*, 2013.

[52] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[53] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, pages 237–246, 2008.

[54] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–108, 2009.

[55] P. Felber, T. Riegel, C. Fetzer, M. Süsskraut, U. Müller, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, 2007.

[56] P. Felber, E. Rivière, W. Moreira, D. Harmanci, P. Marlier, S. Diestelhorst, M. Hohmuth, M. Pohlack, A. Cristal, I. Hur, O. Unsal, P. Stenström, A. Dragojevic, R. Guerraoui, M. Kapalka, V. Gramoli, U. Drepper, S. Tomi?, Y. Afek, G. Korland, N. Shavit, C. Fetzer, M. Nowack, and T. Riegel. The velox transactional memory stack. *IEEE Micro*, 30(5):76–87, Sept 2010.

[57] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.*, 30, 2008.

[58] J. Gibson and V. Gramoli. Why non-blocking operations could be selfish. Technical report, University of Sydney, 2015. http://sydney.edu.au/engineering/it/~gramoli/pubs/contention.pdf.

[59] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, pages 1–10, 2015.

[60] V. Gramoli and R. Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1):86–93, 2014.

[61] V. Gramoli and R. Guerraoui. Reusable concurrent data types. In *ECOOP*, pages 182–206, 2014.

[62] V. Gramoli, R. Guerraoui, and A.-M. Kermarrec. Profiling transactional applications. In *Proceedings of the International Conference on Networked Systems*, 2015.

[63] V. Gramoli, R. Guerraoui, and M. Letia. Composing relaxed transactions. In *IPDPS*, 2013.

[64] V. Gramoli, R. Guerraoui, and V. Trigonakis. TM2C: A software transactional memory for many-cores. In *EuroSys*, pages 351–364, 2012.

[65] V. Gramoli, D. Harmanci, and P. Felber. On the input acceptance of transactional memory. *Parallel Processing Letters (PPL)*, 20(1), 2010.

[66] V. Gramoli, P. Kuznetsov, and S. Ravi. Brief announcement: From sequential to concurrent: Correctness and relative efficiency. In *PODC*, pages 241–242, 2012.

[67] V. Gramoli and A. E. Santosa. Why inheritance anomaly is not worth solving. In *ICOOOLPS*, pages 6:1–6:12, 2014.

[68] D. Grossman. The transactional memory / garbage collection analogy. *SIGPLAN Not.*, 2007.

[69] R. Guerraoui, R. Capobianchi, A. Lanusse, and P. Roux. Nesting actions through asynchronous message passing: the ACS protocol. In *ECOOP*, volume 615 of *LNCS*, pages 170–184, 1992.

[70] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC*, pages 305–319, 2008.

[71] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC*, 2005.

[72] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.

[73] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan&Claypool, 2010.

[74] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *EuroSys*, pages 315–324, 2007.

[75] D. Harmanci, P. Felber, V. Gramoli, and C. Fetzer. TMunit: Testing software transactional memories. In *4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[76] D. Harmanci, V. Gramoli, and P. Felber. Atomic boxes: Coordinated exception handling with transactional memory. In *ECOOP*, volume 6813 of *LNCS*, pages 634–657, Jul 2011.

[77] D. Harmanci, V. Gramoli, P. Felber, and C. Fetzer. Extensible transactional memory testbed. *J. of Parallel and Distributed Computing*, 70(10):1053–1067, March 2010.

[78] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[79] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[80] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, 2005.

[81] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, 2005.

[82] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*, 2006.

[83] S. Heller, M. Herlihy, V. Luchangco, M. Moir, and N. Scherer III, William N.and Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.

[84] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, 1990.

[85] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):123–149, 1991.

[86] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, pages 207–216, 2008.

[87] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.

[88] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[89] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[90] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kauffman, 2008.

[91] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12:463–492, July 1990.

[92] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108–109, 2010.

[93] P. Hyman. In honor of Alan Turing. *Commun. ACM*, 55(9):20–23, 2012.

[94] D. Imbs and M. Raynal. Virtual world consistency: A condition for stm systems (with a versatile protocol with invisible read operations). *Theor. Comput. Sci.*, 444:113–127, July 2012.

[95] Intel. Performance analysis guide - Intel developer zone, 2008. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[96] Intel Corporation. Intel transactional memory compiler and runtime application binary interface, May 2009.

[97] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: a hybrid memory model for accelerators. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 429–440, New York, NY, USA, 2010. ACM.

[98] T. Knight. An architecture for mostly functional languages. In *LFP*, pages 105–112, 1986.

[99] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive software transactional memory. *Transactions on HiPEAC*, 5(2), 2010.

[100] H. F. Korth. Locking primitives in a database system. *J. ACM*, 30(1):55–79, 1983.

[101] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[102] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP*, pages 209–220, 2006.

[103] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *Transact*, 2009.

[104] B. Liskov. The argus language and system. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, volume 190 of *LNCS*, pages 343–430, 1985.

[105] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. In *POPL*, pages 7–19, 1982.

[106] N. A. Lynch. Multilevel atomicity a new correctness criterion for database concurrency control. *ACM Trans. Database Syst.*, 8, 1983.

[107] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *DISC*, 2005.

[108] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *Transact*, 2006.

[109] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *ICPP*, 2008.

[110] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5, 2006.

[111] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.

[112] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research directions in concurrent object-oriented programming*, pages 107–150. MIT Press, 1993.

[113] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for java STM. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, pages 314–325, 2008.

[114] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–64, 1987.

[115] B. Meyer. Systematic concurrent object-oriented programming. *Commun. ACM*, 36(9):56–80, Sept. 1993.

[116] M. Michael. Overview of Power HTM, 2014. Personal communication at the 6th Workshop on the Theory of Transactional Memory.

[117] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[118] J. E. B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, 2006.

[119] Y. Ni, V. Menon, A.-R. Abd-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP*, New York, NY, USA, 2007. ACM.

[120] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA*, 2008.

[121] U. of Glasgow. Scientists squeeze more than 1,000 cores on to computer chip, 2010. http://goo.gl/KdBbW.

[122] V. Pankratius and A. Adl-Tabatabai. Software engineering with transactional memory versus locks in practice. *Theory Comput. Syst.*, 55(3):555–590, 2014.

[123] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[124] A. Reuter. Concurrency on high-traffic data elements. In *PODS*, pages 83–92, 1982.

[125] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298, 2006.

[126] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA*, 2007.

[127] 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*, 2006.

[128] W. Scherer and M. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[129] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.

[130] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, 1984.

[131] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC*, 2007.

[132] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

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

[134] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI*, 2007.

[135] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *PODC*, 2007.

[136] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking transactions without indirection using alert-on-update. In *SPAA*, 2007.

[137] H. Sutter. Choose concurrency-friendly data structures. *Dr. Dobb's Journal*, 33(7), July 2008.

[138] Tilera TILE-Gx. http://www.tilera.com/products/processors/TILE-Gx_Family, 2014.

[139] F. A. Torshizi, J. S. Ostroff, R. F. Paige, and M. Chechik. The SCOOP concurrency model in Java-like languages. In *CPA*, pages 7–24, 2009.

[140] I. L. Traiger. Trends in system aspects of database management. In *ICOD*, pages 1–21, 1983.

[141] Transactional Memory Specification Drafting Group. Draft specification of transactional language constructs for C++, 2009. http://software.intel.com/file/21569.

[142] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *PACT*, pages 127–136, 2012.

[143] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, 2006.

[144] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, 1988.

[145] G. Weikum. A theoretical foundation of multi-level concurrency control. In *PODS*, pages 31–43, 1986.

[146] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[147] S. West, S. Nanz, and B. Meyer. A modular scheme for deadlock prevention in an object-oriented programming model. In *ICFEM*, pages 597–612, 2010.

[148] M. Yannakakis. Serializability by locking. *J. ACM*, 31(2):227–244, 1984.

[149] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA*, pages 265–274, 2008.