Evaluation of Transactional Memory and Other Techniques to Improve the Performance of Algorithms in High Energy Physics for New Processor Architectures

Fachhochschule Münster University of Applied Sciences

Masterarbeit

Autor:	Philipp Schoppe
Fachbereich:	Elektrotechnik und Informatik
Matrikelnummer:	630508
Erstgutachter:	Prof. Dr. rer. nat. Nikolaus Wulff *University of Applied Sciences Muenster*
Zweitgutachter:	Dr. Pere Mato Vila *CERN, Genève, CH*
Datum:	17.11.2014

Abstract

CERN domiciles the Large Hadron Collider, the world's most powerful particle accelerator. Its experiments record vast amounts of measurements that can be analyzed by physicists to study the basic constituents of matter. Expensive algorithms performed on the collected data stress available computing capability, promoting use of multiprocessor systems to accommodate the need for more efficient computation.

When running concurrent software, managing memory access is critical. Hence, Herlihy and Moss introduced Transactional Memory (TM), a multiprocessor architecture encapsulating concurrent operations in database-style transactions. Many TM software implementations have been deemed inefficient, and only by the end of 2013, the first widely available consumer-grade hardware implementation was released to the public by Intel.

This study evaluates several aspects of TM and traditional synchronization techniques. A set of benchmarks is used to experimentally analyze performance, and to investigate additional properties such as scalability and usage from a software engineering perspective.

It is demonstrated that TM can compete with coarse-grained mutual exclusion, and that composable transactions make generic programming possible.

Having hardware support makes TM competitive for various use cases, but as of today, it cannot compete in performance with highly optimized lock-free algorithms and fine-grained locking. Future hardware implementations might improve this situation. An ongoing integration effort into the C++ standard could ease building concurrent software for programmers in the near future, but migrating large software projects is more comprehensive due to the need for additional function annotations.

Acknowledgments

First of all, I would like to express my deepest appreciation to Professor Nikolaus Wulff for supervising this thesis and hinting me into the direction of CERN. I thank him and Professor Peter Mättig for their letters of recommendation for the CERN Technical Student Programme.

Furthermore, I would like to thank my supervisor at CERN, Dr. Pere Mato Vila, for welcoming me into the ROOT team and for giving me the opportunity to work on this exciting field of research. My gratitude goes out to him, Danilo Piparo and Daniel Funke for their suggestions, their valuable advice and our interesting discussions.

I am grateful to all people who proofread this work. I thank Johannes de Fine Licht for his very profound remarks on language, grammar and overall consistency. Daniel Funke helped a lot by indicating parts that had not been sufficiently clear and by checking the theoretical background. Georg Sieber corrected me on physics and supported me while creating my plots. Julia Beuker helped by pointing out repetitive phrases, formal issues and inconsistencies.

I owe my deepest gratitude to my girlfriend Julia for her support and patience during my eight months abroad. Finally, a special thanks to my parents Angelika and Friedrich-Wilhelm without whom none of this would have been possible.

Contents

List of Figures

List of Tables

Listings

Acronyms

ACID Atomicity, Consistency, Isolation, Durability

AMQP Advanced Message Queuing Protocol

ASF Advanced Synchronization Facility

ASIC Application-Specific Integrated Circuit

CERN Organisation Européenne pour la Recherche Nucléaire

DBMS Database Management System

DAQ Data Acquisition

FPGA Field Programmable Gate Array

GPGPU General Purpose Computation on Graphics Processing Unit

HEP High Energy Physics

HLE Hardware Lock Elision

HPC High Performance Computing

IC Integrated Circuit

LHC Large Hadron Collider

MPI Message Passing Interface

NUMA Non-Uniform Memory Access

OOE Out-of-order Execution

RTM Restricted Transactional Memory

STM Software Transactional Memory

STAMP Stanford Transactional Applications for Multi-Processing

TM Transactional Memory

TSX Intel® Transactional Synchronization Extensions

UMA Uniform Memory Access

UDP User Datagram Protocol

1 Introduction

1.1 High Energy Physics at CERN

The Organisation Européenne pour la Recherche Nucléaire (CERN), or European Organization for Nuclear Research, tries to answer fundamental questions on the universe: "What is the universe made of? How did it come to be the way it is?" [65] The name originates from its nuclear physics laboratory established in 1954 by the twelve founding member states[1] in order to understand the basic constituents of matter - the elementary particles. As of 2014, CERN consists of 21 member states, with Israel joining as the first non-European member in 2013.

After discovering the electron, an interest in elementary physics emerged. Over the course of time, more and more particles were discovered with the help of cosmic ray experiments [74]. These new findings and a better understanding of particles, eventually led to the development of high energy particle accelerators, devices that accelerate controlled beams of particles with the help of electromagnetic fields. Accelerators are used to investigate the structure of atomic nuclei, or even to produce new particles, by inducing high energetic collisions between particles or a particles and experimental objects [77]. CERN domiciles the Large Hadron Collider (LHC), the worlds largest particle accelerator with an approximate circumference of 27 kilometers and a design beam energy of 7 TeV[2] [65]. It started operating in 2010 with a beam energy of 3.5 TeV and is the latest addition to the CERN accelerator complex, "a succession of machines that accelerate particles to increasingly higher energies" (see Figure 1.1) where

[1] The founding member states are: Belgium, Denmark, France, Germany, Greece, Italy, the Netherlands, Norway, Sweden, Switzerland, the United Kingdom and Yugoslavia.

[2] The LHC accelerates two beams in opposite directions, so the center-of-mass energy will be 14 TeV.

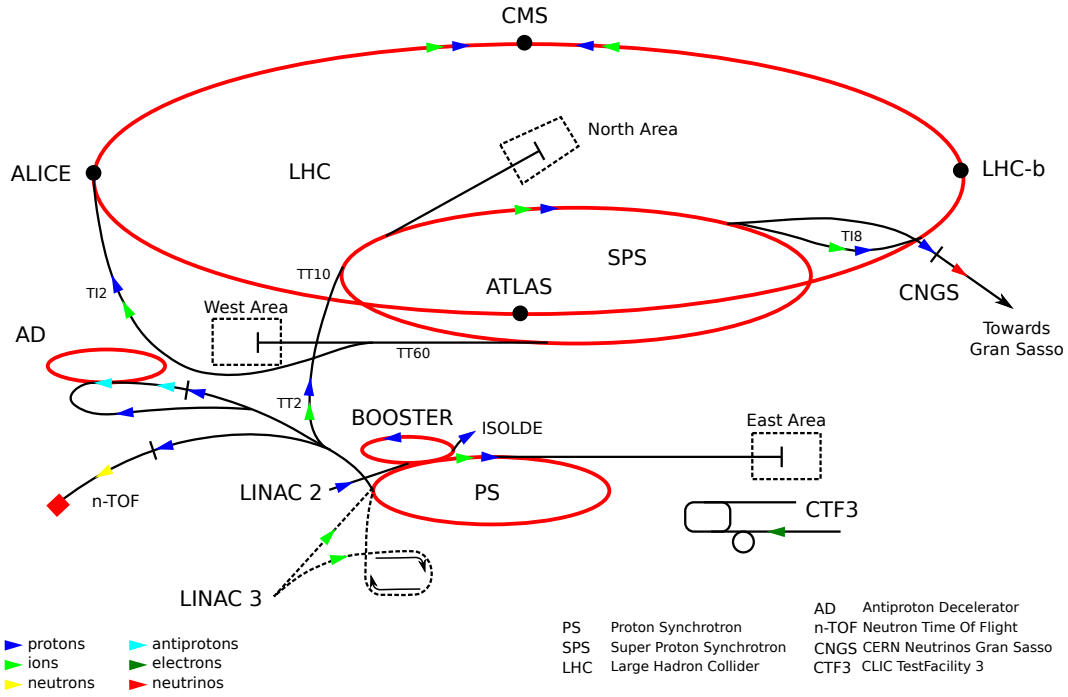

Figure 1.1: The CERN accelerator complex [23].

the older and smaller accelerators act as injection rings for the LHC [15]. The beams are led to collide in a controlled fashion at specific experimental points, the detectors. A detector is a device that is built to observe certain properties of a collision, e.g. a particle's speed, momentum, mass and charge. Particle interactions generate an enormous amount of data: a single event produces up to around one million bytes (1 Mb) of raw data, with the LHC having a beam crossing rate of 40 MHz. The CMS Data Acquisition (DAQ) system is able to read out the detector at a throughput rate of 100GB/s (for LHC Run-1) [20].

Processing, storing and analyzing this amount of data is very challenging. To cope with it, detectors at CERN perform filtering such that only events that potentially show interesting physics processes are processed and stored. Typically, pattern recognition algorithms implemented in customized hardware are used for this purpose[3]. Afterwards, the events are reconstructed and compared to what is predicted by the theory. Part of this reconstruction and analysis process is a simulation of particle collisions in a detector using the GEANT4 [3] and ROOT [10] frameworks, two very compute intensive tasks.

[3] Further information can be found in the technical design report of the TriDAS project [20].

1.2 Motivation

LHC Run-2 is scheduled for early 2015 and will start to run at 6.5 TeV and a reduced beam bunch spacing from 50 ns to 25 ns, resulting in much more data being read out. Computing capabilities reach their limits and the already time consuming calculations will demand even more time. This has created a great need for faster and more efficient algorithms. In the past, the computer industry could rely on Moore's law [71], which made the observation that a CPU's transistor count doubles approximately every two years, which would translate directly into an doubling in clock speed. As thermal limits were reached in Integrated Circuit (IC) development, however, researchers conclude that "the free lunch is over" [88]. Today, instead of rising clock speeds, transistors are spent on parallel hardware architectures. Some of them are briefly mentioned in Section 1.4, but this thesis will focus on multicore processors, processors with two or more independent CPUs (or "cores"). In order to exploit multiple cores, the architectural design changes need to be reflected in software. Naturally, concurrency and parallelism become more and more important in High Performance Computing (HPC), encouraging the High Energy Physics (HEP) community to investigate and implement new data models, software frameworks and algorithms.

When a program operates on multiple cores and the cores share common resources, access to these resources needs to be synchronized. Without a synchronized access, *race conditions* may occur: data gets corrupted by timing dependend *read* and *write* operations that are not properly sequenced. Figure 1.2 illustrates this problem: Two threads try to increment a shared counter variable, but due to unsynchronized modifications, one increment operation is lost. As a solution, the `read` → `modify` → `write` sequence needs to be treated as a single atomic operation. This can either be achieved by an atomic hardware instruction or by using synchronization mechanisms that serialize access to this *critical section*. Note though, that atomic operations are very limited and only few situations can be handled by using them. An overview of several synchronization techniques is given in Chapter 2. All of these techniques have certain properties that introduce new problems or make programming more complex. Nonetheless, they play a critical role in building high-performance and scalable systems.

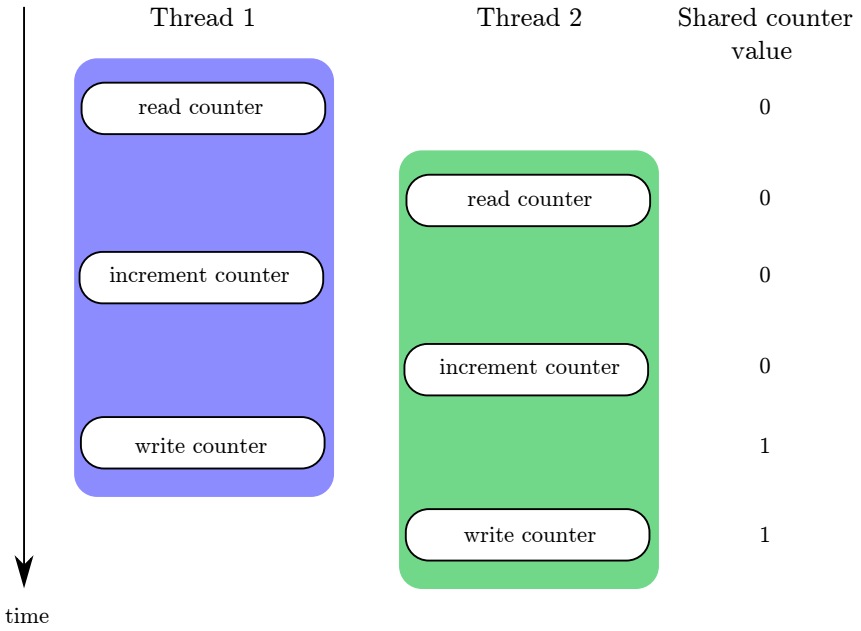

Figure 1.2: Race condition. Two threads try to simultaneously increment a shared counter, but due to unsynchronized modifications one increment operation is lost.

1.3 Goal of This Thesis

The major goal of this thesis is to evaluate performance aspects of several synchronization techniques with a focus on the emerging Transactional Memory (TM) architecture. Other evaluation aspects are properties specific to the mechanisms, scalability and their usage from a software engineering perspective. How error-prone is each technique and how challenging is it to build (efficient) parallel applications when using them? What is the potential of TM and is it feasible to use it in production systems and performance oriented software? All information is targeted at developing with the C++ (C++11 standard) programming language, but the concepts are transferable to other languages, such as Java.

Following this introductory chapter, where Section 1.4 recapitulates prerequisites for performance oriented concurrency, established synchronization techniques are briefly reviewed in Chapter 2 and their defining properties are summarized, before Chapter 3 explains in detail the architectural design of TM. The setup and results of the experimental evaluation study are being presented

in Chapter 4, such that a conclusion can be drawn and an outlook on the future can be made in Chapter 5.

1.4 Prerequisites

Nowadays, CPU clock speeds do not tend to be the limiting performance factor in HPC applications, but rather memory latency becomes more and more critical. Developers must be aware of how memory access (patterns) influence(s) the performance of their programs. Data structure design and data flow becomes a fundamental aspect when building highly concurrent systems. This section outlines some factors that are important to consider when aiming for writing efficient code.

1.4.1 Parallel Computing

Parallel computing architectures exist in various forms. Besides the multiprocessor and multicore architectures, parallel computing can be employed on Field Programmable Gate Arrays (FPGA), General Purpose Computation on Graphics Processing Units (GPGPU), Application-Specific Integrated Circuits (ASIC) or clusters and supercomputers. It always aims for more efficient calculations compared to a sequential solution. The performance gain, or speedup $S(N)$, of having a parallel implementation can be expressed as the ratio between the *best* sequential implementation and the parallel version with N processors:

$$S(N) = \frac{\text{Runtime}_{\text{sequential}}}{\text{Runtime}_{\text{parallel}}(N)} \tag{1.1}$$

In a perfect world, increasing the number of processing units would result in a linear speedup - having two processors instead of one should halve the runtime. In reality though, most problems contain parts that cannot be calculated in parallel and thus prevent a perfect speedup. Their influence regarding the theoretical speedup of a problem has been formulated in Amdahl's law [48]

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{1.2}$$

with N being the number of processors, P being the fraction of a calculation that can be expressed in parallel and $1 - P$ being the fraction that cannot be expressed in parallel. When N tends to infinity, the maximum speedup tends to $\frac{1}{1-P}$. Even small serial fractions descrease the maximum possible speedup heavily. On the other hand, if a program's speedup converges to a limit, a developer could estimate the serial fraction of his program.

As a counterpoint to Amdahl's law - which is very pessimistic regarding the potential speedup through parallelism - Gustafson reevaluated Amdahl's law [42]: The premise of P being independent of N is flawed, since this is almost never the case in practice. Developers do not take a fixed-sized problem and run it on a number of processors, but they scale the problem size to the number of available processors. Gustafson argues that it is most realistic to assume a constant run-time instead of a constant problem size. His refined *scaled speedup* is given by the equation:

$$S(N) = (1 - P) + N \cdot P \tag{1.3}$$

Estimating P is not a trivial exercise in a lot of situations, since the code may be too complex, or calls to external modules may be hard to be analyzed precisely. An alternative approach could be to determine a program's parallelism through observed speedup instead. Such a model is proposed by Allen B. Downey in his paper "A model for speedup of parallel programs" [28]. Based on observed speedup curves, two parameters can be estimated: the average parallelism of a program and its variance. The model describes the relationship between a number of cores and execution time and has proven to reflect the behavior of real programs. Downey describes two use-cases where this comes in handy:

- **Modeling parallel workloads:** simulation tools can use this speedup model to generate stochastic workloads.

- **Summarizing program behavior:** if a program ran on a range of cluster sizes, the program's behavior over time can be captured and used for future scheduling and allocation optimization.

Existing speedup models that are derived from observed speedups have been proposed before. For example, Chiang et al. [18] propose a speedup model

$$S(N) = (1 + \beta)N/(N + \beta) \tag{1.4}$$

with β as the only free parameter being ranged from 0 for a sequential program to infinity for linear speedup. Downey criticizes that this model does not provide clear semantics as to what the parameter β represents in an actual program and thus came up with a model based on the average level of parallelism A and its variance σ. He presents a model for a program with low variance in degree of parallelism ($\sigma \leq 1$), and a model for high variance ($\sigma \geq 1$). The low variance model is defined as

$$S(N) = \begin{cases} \frac{A \cdot N}{A + \sigma/2 \cdot (N-1)} & 1 \leq N \leq A \\ \frac{A \cdot N}{\sigma \cdot (A - 1/2) + N \cdot (1 - \sigma/2)} & A \leq N \leq 2 \cdot A - 1 \\ A & N \geq 2 \cdot A - 1 \end{cases} \tag{1.5}$$

and the high variance model is defined as

$$S(N) = \begin{cases} \frac{N \cdot A \cdot (\sigma+1)}{\sigma \cdot (N + A - 1) + A} & 1 \leq N \leq A + A \cdot \sigma - \sigma \\ A & N \geq A + A \cdot \sigma - \sigma \end{cases} \tag{1.6}$$

Given a set of observed speedups s_i for a number of cores n_i, the parameters A and σ that minimize the sum of squared differences between observed and fitted values can be estimated by minimizing the equation

$$\chi^2(A, \sigma) = \sum_i [s_i - s(n_i; A, \sigma)]^2 \tag{1.7}$$

where $s(n_i; A, \sigma)$ is the modeled speedup calculated by Equation 1.5 and 1.6. This way, a fitting speedup model that reflects the level of parallelism of a program can be calculated.

1.4.2 Multiprocessor Systems

HPC systems make extensive use of multiprocessor platforms, which are the main architectures this thesis contemplates. They perform parallel calculations

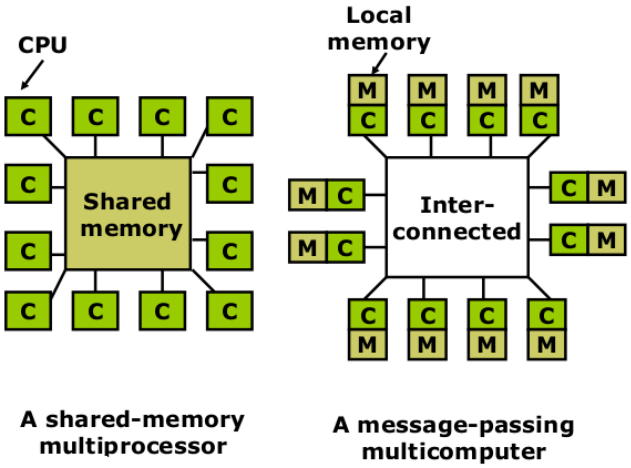

Figure 1.3: UMA and NUMA multiprocessor architectures [41].

either on shared memory segments or synchronize using a Message Passing Interface (MPI) [35]. Figure 1.3 presents the conceptional design of two different multiprocessor systems.

A shared-memory multiprocessor contains multiple cores operating on a single memory segment, thus having no extra communication cost to interchange partial results between different cores. Literature associates this kind of architecture to the Uniform Memory Access (UMA) concept. The memory latency is independent of which processor tries to access memory. Instead, in a message-passing multiprocessor system, all CPUs perform their calculations on private memory, but they need to propagate the results to the other processors by sending messages across the system. In this Non-Uniform Memory Access (NUMA) architecture, the memory latency is depending on the processor which tries to access data in memory. Shared-memory multiprocessors are not free of communication, though. They need concurrency control mechanisms to access shared memory segments in an ordered way, because data races might occur otherwise.

1.4.3 Memory and Caches

CPUs and their registers run at very high speed. Main memory access time is orders of magnitude slower than fetching data from on-chip registers. Since fast memory units are costly, a hierarchical memory model (see Figure 1.4) is

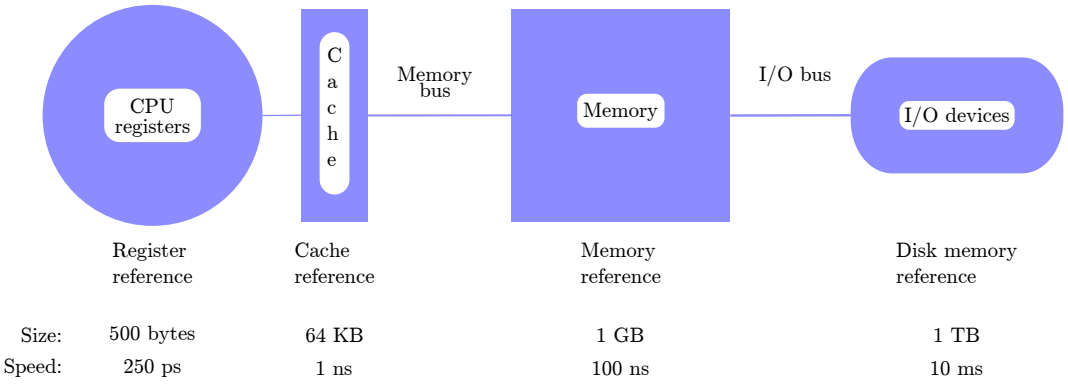

Figure 1.4: Memory hierarchy. The levels in a typical memory hierarchy in embedded, desktop, and server computers. As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by factors of 10 - from picoseconds to milliseconds - and that the size units change by factors of 1000 - from bytes to terabytes [48].

used. As already mentioned, memory access is becoming a critical bottleneck in application performance. To overcome this bottleneck, high-speed memory buffers (namely the L1 and L2 cache) are used. When data from the main memory is requested by the CPU, retrieved data is stored in the cache and can be accessed faster if needed multiple times. Data from the main memory is fetched into lines (or blocks). Those lines will be referred to as *cache lines*. Since the L1 and L2 caches are small, only a limited amount of data can be cached. Today, a common cache line size is 64 bytes. Manufacturers could build bigger cache memory, but it is not economical [31]. Lines already stored in the buffer constantly get overwritten by new data. If a requested line is not in cache and has to be fetched from the main memory, a so-called *cache miss* occurs. If a requested line already resides in cache, a *cache hit* occurs. A lot of cache misses obviously decrease performance, since slower memory has to be accessed in order to fetch new data.

When multiple threads access data that is cached on the same line, cache systems may protect that line by a hardware write lock that can only be owned by one core at a time [90]. Additionally, changes to the cache need to be made known to all processors of a system. This is ensured by the *cache coherence protocol*, a protocol which communicates all changes to the cores through a

system's northbridge[4] and ensures a consistent state of memory. Consequently, it may happen that multiple cores constantly invalidate a single shared cache line, forcing the cache coherence protocol to propagate memory changes over the bus and reload new data from the slow main memory - which decreases performance drastically. This phenomenon is called *false sharing*, a problem that due to its subtlety sometimes is not accounted for by programmers. Listing 1.1 illustrates how this problem may be triggered in code [91].

Listing 1.1: False sharing example code.

```
struct container {
    int x;
    int y;
};

static struct container f;

Thread 1:

int sum_a() {
    int s = 0;

    for (int i = 0; i < 1000000; ++i) {
        s += f.x;
    }

    return s;
}

Thread 2:

void inc_b() {
    for (int i = 0; i < 1000000; ++i) {
        ++f.y;
    }
}
```

In this example, the `sum_a` and `inc_b` functions run concurrently and each access one member of `struct container`. Two `int` types are most likely stored on the same cache line. When one function accesses the member variable and modifies its value, the other function probably already did the same and invalidated the cache line. The new value has to be retrieved from main memory and the variable is then modified. Now the cache line is invalidated again for the other executing function. Although the two functions are logically separated,

[4] A microchip directly connected to the CPU that handles high performance communication among CPUs.

they still affect each other. Literature sometimes refers to this problem as *cache ping-pong*.

1.4.4 Memory Model

When a programmer writes code, the statements are written in a logical order - the *program order* - to reflect a certain logic or algorithm. Modern microprocessors and compilers are heavily optimized to actively rearrange instructions in order to reduce or hide latency that occurs due to varying memory access times. This is done without changing a program's logic: compilers may eliminate expressions, control the allocation of registers or perform loop unrolling. Processors make use of Out-of-order Execution (OOE), a dynamic technique that alters the instruction order in a CPU's pipeline during runtime. The order of operations that is written in source code is not identical to what is actually executed, though its logic is preserved - *sequential consistency* is guaranteed [51].

These optimizations work perfectly fine for the serial use case or when threads only read from shared resources and merely modify thread local data. As soon as threads start to modify shared memory locations, race conditions occur. To avoid this scenario, sequential ordering has to be enforced, since a reordering of operations may change the logic in a multi-threaded application. A compiler can only reason about the single-threaded scenario if critical regions are not explicitly handled. Listing 1.2 illustrates how OOE can affect the logic of a multi-threaded program.

Listing 1.2: Out-of-order execution affects execution.

```
Initialization:
    bool gPrintFlag  = false;
    int  gPrintValue = 0;

Thread 1:
    gPrintValue = 2014;
    gPrintFlag  = true;

Thread 2:
    while (gPrintFlag == false) { }

    std::cout <<  gPrintValue << std::endl;
```

The idea of the application is to print out the value of the variable `gPrintValue` (2014), if the global boolean variable `gPrintFlag` has been set to `true` by the first thread. In the sequential case it would be perfectly fine to rearrange the variable assignments in thread 1, such that `gPrintFlag` is set to `true` before `gPrintValue` is set to 2014. When running multiple threads, the order must not be changed, otherwise `gPrintFlag` is set to `true` before `gPrintValue` has been set up as intended.

In a multi-threaded application, ordering semantics can be enforced by different kinds of *memory barriers* or *fences* that guarantee that changes to shared resources are globally being made visible to other threads at a specified point [89]:

- **acquire barrier:** prevent read and write operations behind this barrier to be reordered before it.
- **release barrier:** prevent read and write operations before this barrier to be reordered behind it.
- **full barrier:** prevent reordering of read and write operations in either direction

These requirements are reflected in a programming language by establishing a *memory model*, a specification that allows developers to reason about memory layout and sequential consistency regardless of the underlying compiler or platform, and make low level primitives for multi-threading available. The C++11 standard first introduced such a model to the C++ language and also extends previous standards by adding multi-threading primitives support. When dealing with atomic operations, a full barrier for sequential memory ordering is assumed by default, if not specified otherwise. C++11 provides six different memory ordering specifiers that regulate how non-atomic memory accesses are ordered around atomic operations, with `memory_order_seq_cst` enforcing the default sequential order. Understanding the memory model with its details is extremely difficult and most programmers will not need to do so. Leading C++ experts even discourage developers from using it directly [89].

2 Concurrency Control

Managing shared resources is critical when exploiting concurrency in applications. *Concurrency control* techniques are means to provide a form of data protection, by ensuring an ordered access to shared data when a single (or sequence of independent) atomic operation is not sufficient. This chapter illustrates several well known synchronization concepts and primitives.

2.1 Atomic Hardware Instructions

Before presenting some of the prevailing concurrency control techniques in Section 2.2, some hardware supported atomic instructions are listed. They are non-interruptable and are the building blocks to efficient high-level synchronization mechanisms.

- `test-and-set`: write to a memory location and return its old value as a single atomic operation.
- `atomic-increment`: load the current from memory to a register, increment it, and write it back to memory (see Figure 1.2).
- `compare-and-swap`: compare two memory locations to a given value. If they are equal, assign a given memory location to a new value.
- `load-link/store-conditional`: return a memory location's value and store a new value into it, if no updates have occured since the load.

2.2 Traditional Approaches

2.2.1 Mutual Exclusion

Mutual exclusion was first introduced by Edsger W. Dijkstra [27] in 1965. In concurrent programming, a critical section of code accessing shared resources should only be executed by one process at a time. One way of ensuring this requirement is through *locking* of this section. Today, this is a very common and probably the most dominant approach to concurrency control.

Mutex

A mutex is a synchronization primitive provided by the operating system that a process can atomically gain ownership of. Only one process may acquire the mutex at any given time, and all other processes must wait until the owner of the lock releases it again. Typically, a process *blocks* (and possibly is put to sleep by the operating system's scheduler) until it successfully acquires the mutex. Mutual exclusion can be achieved if all processes that execute a program agree that each process acquires ownership of a mutex before entering its critical region, and then releases it again after leaving the region. This way, only one process at a time may modify shared data. In C++11, a critical section can be protected by a mutex as shown in Listing 2.1.

Listing 2.1: Mutual exclusion.

```
std::mutex g_mutex;

int g_data;

void increment_data() {
    g_mutex.lock();
    ++g_data;
    g_mutex.unlock();
}
```

A mutex - and mutual exclusion in general - is not an ideal solution regarding concurrency control, since it comes with a set of drawbacks:

- **Deadlock:** a deadlock is a situation where processes lock a set of objects with two or more mutexes and they each wait for the lock owned by

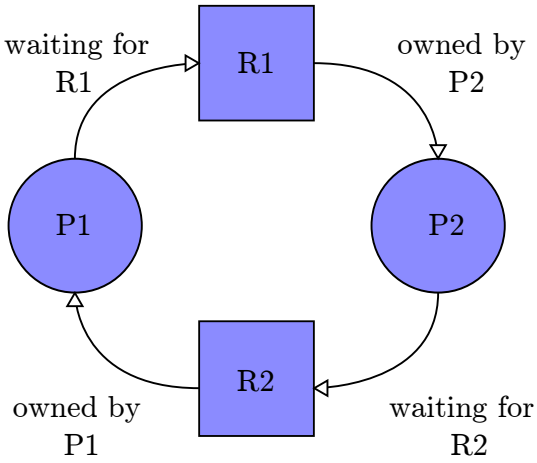

Figure 2.1: A deadlock situation: Process $P1$ waits for the resource $R1$ which is currently owned by a process $P2$. Meanwhile, $P2$ waits for the resource $R2$ which is owned by $P1$ [99].

another process. No process can continue to work, as Figure 2.1 illustrates. Deadlock situations can be very hard to detect before runtime, since they may be timing dependent. They are a very serious issue in concurrency programming, and might even be induced externally through library calls.

- **Priority inversion:** a low priority process may hold a lock that is needed by a high priority process, effectively disabling priority scheduling.

- **Convoying:** a process may be descheduled or interrupted while holding a lock. Consequently, all other processes have to wait for it to be rescheduled.

When designing for high performance applications, a programmer needs to tune his data structures for fine-grained locking to reduce waiting times as much as possible. Often, multiple locks have to be placed around several critical sections, making it very difficult to develop an error-free program. Designing efficient concurrent software architectures is a very complex task, and the solution to deadlock-free algorithms is not always obvious, as illustrated by the "dining philosophers problem" [52]. A software designer may even decide that it is best to anticipate deadlock situations. In doing so, he could foresee a deadlock detection and recovery algorithm that allows deadlock situations to occur, but is able to resolve them [83].

Reentrant Mutex

Reentrant mutexes can be acquired multiple times by the same process (and has to be released the same amount of times again). When locking a normal mutex more than once, the process would deadlock. This allows to easily acquire a lock in recursive algorithms.

Spinlock

A spinlock is a lock that provides mutual exclusion functionality, but instead of blocking when ownership of the lock could not be acquired, it will wait for it by constantly retrying ("spinning" on the lock). It can be implemented using an atomic `test-and-set` operation that exclusively sets and unsets a shared flag. For small critical sections, spinlocks can achieve a higher performance than blocking mutexes [5]. The C++11 standard does not provide a spinlock primitive (the pthread library does), but it can be implemented using an atomic flag, as Listing 2.2 demonstrates.

Listing 2.2: Spinlock functionality using C++11 atomics.

```
#include <atomic>

class SpinLock
{
public:
    void lock()
    {
        while(lck.test_and_set(std::memory_order_acquire))
        {}
    }

    void unlock()
    {
        lck.clear(std::memory_order_release);
    }

private:
    std::atomic_flag lck = ATOMIC_FLAG_INIT;
};
```

Readers-Writer Lock

A readers-writer lock (or a read write mutex) is a locking and signaling primitive that distinguishes between being acquired for read or write access and is intended as a solution to the *readers-writers problem* [24]. This comes in handy

in situations where multiple processes read from a shared resource, while others modify it. Suppose process A attempts to read from the resource and therefore locks it through a mutex. This is very restrictive, because another process B who also wants to read the data has to wait for process A to finish, even though its read operation would be non-intrusive to the shared data. A readers-writer lock can be used to allow the reading process B to directly enter its critical section. When the last active reading process finishes its operation, processes that are waiting in order to write to the resource are notified that they may start doing so. However, this can lead to *starvation* of a writer process, since it is possible that it has to wait forever when there are readers at all times. Thus this solution has a *readers-preference*. Another implementation could implement a *writers-preference*, a mechanism that ensures that no writer has to wait longer than necessary: already reading processes are allowed to finish their operations, but no new readers are allowed to enter the critical section. Readers-writer locks can be implemented based on mutexes and condition variables or semaphores (two synchronization primitives that provide a signaling mechanism).

2.2.2 Lock and Wait-Free Datastructures

Mutual exclusion is based on blocking an active process, if necessary. Incorrect usage of mutexes may cause deadlocks that are difficult to detect and only appear in rare timing dependent scenarios. Leaving deadlocks aside, blocking data structures or algorithms have the effect of slow, or even halted processes, possibly preventing faster processes from running at full speed [70]. Processes might be delayed by cache misses, scheduling and priorities or page faults. Another approach to concurrency control that possibly avoids these issues is the class of *non-blocking* algorithms and data structures. They allow concurrency without using blocking synchronization mechanisms. Two important categories of non-blocking data structures are *lock-free*[5] and *wait-free* data structures. Herlihy gives a definition of these properties in [49].

[5] Non-blocking is often used as a synonym for lock-free which is technically incorrect. The spinlock shown in Listing 2.2 does not use blocking calls, but it still *locks* a critical region that is only accessible by one process at a time. Threads owning a spinlock might be preempted by the operating system which violates lock-freedom.

Lock-free. *A concurrent data structure is* lock-free *if a process is guaranteed to complete an operation on it after the system as a whole takes a finite number of steps.*

Wait-free. *A concurrent data structure is* wait-free*, if each process is guaranteed to complete an operation on it after taking a finite number of steps.*

Lock-freedom ensures overall program progress, whereas wait-freedom ensures progress in a bounded number of steps. The latter property is very desirable in real-time systems, or for interrupt handlers which must not block. Since lock and wait-free algorithms are very complex, not many practical implementations do exist [64]. While lock-free algorithms have been subject to research for years, only few efficient and correct implementations are known. Lock-free research could only successfully be applied to a very limited range of data structures and having a working algorithm is almost always a publishable result, with queues being the most dominating data structures in this field of research. Hence, lock-freedom is only an option for production systems in a limited amount of use-cases. Even if lock-freedom is achieved for a specific task, the algorithm might perform very poorly. Lock-freedom does not inherently grant good performance.

Non-blocking functionality is almost always achieved with the help of atomic `read-modify-write` instructions, such as `compare-and-swap` (CAS), `fetch-and-add` or `test-and-set`. To use them efficiently, hardware support is mandatory (which is the case on all modern architectures). In C++11, `std::atomic<>` objects can be inspected through their member function `is_lock_free()` in order to see whether they are truly lock-free or whether they are implemented in a different (fallback) way on the underlying hardware platform. This class of instructions typically operates on word sized data. Therefore, lock-freedom using this set of instructions is typically designed to modify pointers such that changes to shared resources are globally made known to the system through a single pointer update. Atomic increments of integers may be sufficient in other use cases, e.g. when managing memory in arrays and accessing data through indices. The wait-freedom property is often achieved by having processes help each other in order to complete their operations. Such a helper mechanism could be implemented through announcing the next operation in a shared state array, see Kogan's report on wait-free queues [63]

as a reference. But wait-free data structures tend to be inefficient in practice due to this kind of helping functionality. Kogan and Petrank [64] reduced the overhead by introducing a *fast-path-slow-path* mechanism. The *fast-path*ensures good performance, but after a certain amount of time, a slow path is chosen in order to guarantee the wait-freedom property. In the majority of cases, the fast path is sufficient and completes the operation, and only in a few situations is slow path actually chosen.

Lock-free programming is very complex, since there are no critical sections that serialize shared data access when entered. For every single instruction, a developer must keep in mind that other threads might have changed shared data and that references might have changed or objects have been deleted. Memory management is essential - deleting objects must be avoided when they are possibly referenced by other threads, but the memory footprint has to be kept as low as possible. Dynamic memory management is a fundamental challenge in lock-free programming. This is an area where garbage collections can be of great help. In his book "Concurrency in Action" [92], Anthony Williams lists three techniques that help tackling this problem:

- Waiting until no threads are accessing the data structure and deleting all objects that are pending deletion.
- Using *hazard pointers* to identify that a thread is accessing a particular object.
- Reference counting the objects so that they are not deleted until there are no outstanding references.

Hazard pointers are a methodology for memory reclamation for lock-free (and wait-free) dynamic objects presented by Maged Michael [69]. A thread that is accessing a possibly shared object first sets a hazard pointer to reference it. In doing so, other threads know that deleting this object is "hazardous". If the thread does not use the referenced object anymore, the pointer is cleared.

Another related difficulty in lock-free programming is coping with the *ABA problem*: a thread reads a value A (e.g. a pointer pointing to an allocated object) and is then preempted by the operating system scheduler. In the meantime, another thread then changes this value to B. It changes the associated object

(maybe even frees the object) and performs some operation that than sets the value back to A (a new object may have been allocated at the old position). The original thread proceeds, incorrectly assuming that the value is still the same, and thus operates on malicious data.

Since lock-freedom does play an important role regarding TM, Listing A.1 in the appendix shows a lock-free queue implementation by Michael and Scott which is widely regarded as one of the fastest (if not the fastest) lock-free implementation of a queue. The code is taken from their original publication in 1996 [70]. It has been formally validated by [39].

2.2.3 Issues with Traditional Concurrency Control

Pieter Hintjens, main designer of the Advanced Message Queuing Protocol (AMQP), and Martin Sustrik, architect and maintainer of *ZeroMQ*, discuss modern multithreading in their article "Multithreading Magic" and see a lot of issues with "The Painful State of the Art". Based on a technical article published by Microsoft ("Solving 11 Likely Problems in Your Multithreaded Code"), they describe a lot of drawbacks with the traditional approach to concurrency. Some of them have already been addressed in previous sections, but to further outline the need for new concepts and techniques, an overview is given:

- **Forgotten Synchronization:** forgotten or not correctly implemented synchronization causes race conditions and deadlocks. Sometimes, errors may only be noticed when eventually appearing in production code.

- **Incorrect granularity:** even when correctly defining logical critical sections, issues may arise. Critical sections may be too big and slow down other threads.

- **Read and Write Tearing:** modifying 32-bit or 64-bit values on naturally sized words (4 bytes on 32 bit architectures, 8 bytes on 64 bit architectures) is atomic and synchronization can sometimes be skipped. But when a variable is not properly aligned or not naturally sized, read or write tearing appears.

- **Lock-free reordering:** memory barriers need to be setup in order to cope with OOE.
- **Lock Convoys:** many threads trying to repeatedly acquire lock ownership may effectively halt an application.
- **Two-step Dance:** threads bounce between waking and waiting.
- **Priority Inversion:** a lower priority thread blocks threads with high priority.

Another important aspect to concurrent software is *contention*. Contention is a conflict between multiple threads that simoultaneously want to acess a shared resource, try to acquire a lock, or want to perform an atomic operation. The more threads are actively accessing a shared resource, the more threads have to wait for their turn and the total system progress is delayed.

From a software engineering point of view, developing multi-threaded software is orders of magnitude more complicated than developing non concurrent software. Based on their own experience, Hintjens and Sustrik estimate an increase in costs of at least a factor 10 for developing and maintaining concurrent software. Furthermore, traditional approaches will often not scale beyond a limited number of threads. Some code can be made concurrent, but achieving good scalability is very cumbersome and in many cases not possible at all. Scaling an application for more than about 10 threads often requires completely lock-free code. As already stated, developing lock-free code is very complex. Two alternatives to the traditional approaches are *message-passing* and *transactional* programming models. They are being introduced in the next two sections. An additional complexity is debugging: muli-threaded applications may have non-deterministic behavior due to varying timings and it is difficult to understand certain failures and to reproduce error states.

2.3 Message Passing Interface

In a message-passing parallel programming model, data and information is sent to other processes through messages, similar to network protocol communication

over User Datagram Protocol (UDP). Threads only process private data, thus having no need for the classic synchronization techniques. This makes programming a lot simpler and reduces programming complexity. When synchronization of temporary results is needed, cooperating processes may perform this by making their state and data known through messages. Depending on the data layout and problem domain, very scalable applications can be implemented. In an ideal scenario - perhaps when working with *embarrassingly parallel data* (data on which calculations can be performed without communication costs) - this approach would scale up to an infinite amount of processors.

MPI is a message-passing library interface specification by the MPI Forum [35] that describes a message-passing programming model. Programmers who are familiar with the BSD socket library API will not need a long introduction to start using it. The well known *ZeroMQ* and *nanomsg* do not implement the MPI, but also perform application level concurrency via message passing. The authors of ZeroMQ report that this approach has several advantages over lock based synchronization: the application never blocks and all the CPU time can be dedicated to the actual work [53]. Message-passing systems can be setup between multiple cores on the same machine, but also in complex grids that are made of hundreds or thousands of machines that pass messages over the network, or smaller HPC clusters connected via InfiniBand [8].

2.4 Transactions

The transaction paradigm is a well established concept and famous for being used in Database Management Systems (DBMS). Transactions encapsulate a sequence of operations into a single operation that then either *commits* the result of a modifiction, addition or deletion into the database, or *aborts* in case of an error. If the transation is unsuccessful, the operation can retry the commit. A transaction can be seen as a unit of work - or as associated operations - that can be executed concurrently without compromising the underlying database's integrity. In order to achieve this, transactions in a modern DBMS are required to fulfill a set of properties calledAtomicity, Consistency, Isolation, Durability (ACID). Based on the work of Eswaran et al. "The Notions of

Consistency and Predicate Locks in a Database System" [32], Haerder and Reuter [43] describe this important set of properties as follows:

- **Atomicity:** either all operations take effect, or nothing happens.
- **Consistency:** a transaction can only commit legal results, leaving the system in a valid state.
- **Isolation:** operations within a transaction are hidden from other, concurrently running transactions.
- **Durability:** when successfully committing, a transaction's changes are guaranteed to be permanent.

If a transaction fails while executing, the system has to perform a *roll-back* where it restores the original database state. Transactions are a conceptual design of concurrency; their internal implementation is hidden from the users and most likely based on one of the previously mentioned mechanisms. Alternatively, a DBMS can rely on a scheduler that serializes all transactions.

Chapter 3 will introduce TM, a transaction based architecture for multiprocessor systems. TM is inspired by the database style transactions and offers easy to use lock-freedom as an alternative to mutual exclusion-based synchronization.

3 Transactional Memory

In 1993 Maurice Herlihy and J. Elliot B. Moss introduced a new multiprocessor architecture named Transactional Memory (TM) [50]. Its purpose is to provide means to efficient lock-free synchronization easily accessible by programmers. Operations that modify critical data are encapsulated in atomic transactions, thus avoiding the need for mutual exclusion. The concept of TM became a hot research topic and it evolved into a promising programming idiom with several software libraries being popularized under the context of Software Transactional Memory (STM). Only in the end of 2013, Intel released the first hardware implementation that was widely available to the public consumer market.

3.1 History

Hardware architectures that support a transactional synchronization mechanism have already been investigated in the 1980s: In 1986 Tom Knight proposed parallel transactional blocks for "mostly functional" languages based on exploiting the cache coherence protocol [62]. Chang and Mergen presented a database-style transaction concept, which they implemented in the IBM 801 storage system [16] in 1987. Following this, Maurice Herlihy and J. Elliot B. Moss published the famous paper: "Transactional Memory: Architectural Support for Lock-Free Data Structures" [50]. Two years later, Shavit and Touitou presented the idea of STM [81]. These two papers mark the beginning of TM.

Over the course of time, a lot of research effort was put into TM which led to many ideas on various levels of abstractions, including programming language specifications [93], STM algorithms and implementations [2, 11, 33, 87], system

library and compiler support [54, 79, 84], hardware implementations [19, 57, 58] and kernel scheduling algorithms [67].

While many different STM libraries for a wide range of programming languages are available, the history of actual hardware implementations is very brief. The first commercial processor to support TM is Sun's *Rock*, a high-performance SPARC CMT processor [17], but it was never released, and only prototypes were given to researchers [59]. In 2009 AMD proposed Advanced Synchronization Facility (ASF), "an AMD64 extension to allow user- and system-level code to modify a set of memory objects atomically without requiring expensive traditional synchronization mechanisms" [1], though it is still in its proposal stage and has yet to be released in hardware. IBM's Blue Gene/Q processor [45] supports TM and was first deployed in 2012 as part of the Sequoia supercomputing system in the Lawrence Livermore National Laboratory [9]. Only in the end of 2013, Intel released the first hardware implementation - TSX [57] - that is widely available for the public consumer market. It is implemented on the Intel Haswell processor microarchitecture.

3.2 Architectural Design

3.2.1 Foundations

Herlihy's and Moss' paper title "Transactional Memory: Architectural Support for Lock-Free Data Structures" already suggests the motivation behind TM: Building a lock-free synchronization technique based on transactions that modify memory regions in an atomic fashion. The basic idea is to allow programs to read and modify disjunctive memory locations atomically, just like a database transaction operates on records (excluding the *durability* property of the ACID property set, since data will not be saved when a program terminates). Instead of a DBMS that resolves conflicting transactions, the underlying STM library or hardware is responsible for this task. A transaction in the sense of TM is a composition of (tracked) `LOAD` and `STORE` operations. Section 3.2.2.1 and 3.2.2.2 explain how these instructions can be used in order to implement

the transactions and how they ensure lock-freedom. Transactions can either successfully *commit*, or they *abort* in the case of an error:

1. Commit: a successful commit implies that the concurrent transactions did not interleave with each other (they did not modify the same shared memory resources). All changes can be applied without invoking any form of mutual exclusion and unnecessary synchronization is eliminated.

2. Abort: some concurrent transactions did interleave with each other (they did modify the same shared resource). Affected transactions must not apply their modifications, in fact they need to perform a roll-back to the initial state instead and then retry. This may follow a scheduling or back-off strategy to ensure forward progress.

TM is best suited to replace short critical code sections, since the chance of interfering transactions leading to an abort is smaller. A programmer is free to specify *nested* transactional regions - transactions within transactions - a task that would otherwise risk deadlocks when relying on mutex based synchronization. Executing the transactions is the responsibility of the underlying TM architecture. Therefore, TM achieves a major advantage over other synchronization techniques: The programmer does not need to concern himself with *how* the complexities of correctly executing concurrent operations are handled, but only with *what* needs to be run atomically. In code, this is typically done by explicitly declaring the beginning and end of a transactional region. Therefore, lock-free algorithms that used to be very complex and error-prone become easily available.

Ideally, transactions should be *unbounded*, i.e. being of arbitrary size and duration [4]. STM can manage to do so, but comes with an overhead that can drastically reduce performance, leading to researchers deeming STM to just be a “research toy” [13]. Hardware support could lower the overhead, but chip manufacturers would need to fundamentally redesign parts of their IC layout, a cost that is unreasonably high. An alternative approach is to only support *bounded* transactions - transactions that are limited in size, duration and in types of supported operations. By allowing hardware transactions to fail, and rely on software to resolve issues by providing a fall-back mechanism, hardware requirements decrease. Therefore, bounded transactions could be realized by

extending existing components, e.g. the cache-coherence protocol. This *best effort* approach has been proven useful and appears to be a sound transition concept towards true unbound transactions.

3.2.2 Hardware Transactional Memory

3.2.2.1 Instructions

This section will focus on TM being implemented as an extension to the cache-coherence protocol, since it is the basic building block for the TSX platform. For this approach Herlihy and Moss [50] define the following set of instructions for accessing memory:

- *Load-transactional* (`LT`) reads the value of a shared memory location into a private register.
- *Load-transactional-exclusive* (`LTX`) reads the value of a shared memory location into a private register, "hinting" that the location is likely to be updated.
- *Store-transaction* (`ST`) tentatively writes a value from a private register to a shared memory location. This new value does not become visible to other processors until the transaction successfully commits (see transaction state instructions in Section 3.2.1).

Conceptually, the memory is divided into three logical regions: *read*, *write* and *data set*. Regions that are accessed by the `LT` instruction fall into the category of the read set. `LTX` and `ST` modify the write set. The data set is defined as the union of the read and write sets. TSX reduce this logic to only having a read and write set. A transaction's state can be modified by applying the following instructions [50]:

- *Commit* (`COMMIT`) attempts to make the transaction's tentative changes permanent. It *succeeds* only if no other transaction has updated any location in the transaction's data set, and no other transaction has read any location in this transaction's write set. If it succeeds, the transaction's

changes to its write set become visible to other processes. If it fails, all changes to the write set are discarded. Either way, `COMMIT` returns an indication of success or failure.

- *Abort* (`ABORT`) discards all updates to the write set.

- *Validate* (`VALIDATE`) tests the current transaction status. A *successful* `VALIDATE` returns *True*, indicating that the current transaction has not aborted (although it may do so later). An *unsuccessful* `VALIDATE` returns *False*, indicating that the current transaction has aborted, and discards the transaction's tentative updates.

A possible instruction sequence that replaces a lock around a critical section could be: `LT` → `VALIDATE` → `ST` → `COMMIT`. If the `VALIDATE` or `COMMIT` command indicates failure, the process typically retries the sequence. In a more complex scenario, back-off strategies may be better suited than a simple retry, as Anderson shows while evaluating spin lock performance [5].

When a transaction succeeds, its changes have been made globally visible to other processes without explicit synchronization. The underlying processor *optimistically* executed the transaction, avoiding unnecessary synchronization and serialization and thus ensures lock-freedom. Only if a transaction fails, overhead by the roll-back mechanisms kicks in.

3.2.2.2 Conflict Detection

Conflict detection is implemented by extending the cache coherence protocol: New tags and states are associated with the cache lines, and memory regions of a transaction can be invalidated. Since the TSX specification does not specify details of its implementation, the following paragraph outlines the proposed architecture given by Herlihy and Moss. It is most likely that TSX are implemented in a similar way, since previous chip generations rely on the MESIF protocol and a complete rewrite of the cache coherence protocol is very unlikely. Additionally, TSX operate at a cache line size granularity which reinforces this assumption.

Besides the normal cache, each processor keeps an extra transactional cache for transactional operations. Entries may only be added into one of them - it is impossible for an entry to be in both caches. The transactional cache can be understood as an extra buffer that all active transactions operate on. In the case of an abort, all cache lines that logically reside in the *write set* are invalidated. If a transaction succeeds, its cache lines may be snooped by the system's processors and can be written back into the normal cache. Goodman's snoopy protocol [38] requires each cache line to have two bits defining four states for the residing data:

Name	Access	Shared?	Modified?
INVALID	none	–	–
VALID	R	Yes	No
DIRTY	R, W	No	Yes
RESERVED	R, W	No	No

Table 3.1: Cache line states [50].

The transactional cache adds a tag (listed in Table 3.2) to each state.

Name	Meaning
EMPTY	contains no data
NORMAL	contains commited data
XCOMMIT	discard on commit
XABORT	discard on abort

Table 3.2: Transactional tags [50].

An active transaction caches two entries, one tagged as XCOMMIT and one tagged as XABORT. ST operations only operate on the XABORT entry, leaving the XCOMMIT entry unchanged. If a transaction commits successfully, its XCOMMIT entries are marked as EMPTY - XABORT entries as NORMAL. If a transaction aborts, the entries are changed to EMPTY and NORMAL, respectively. When adding a new entry to the cache, a transaction searches entries in a hierarchical fashion (in descending priority order): EMPTY, NORMAL, XCOMMIT. The entry is then overwritten with the new date. If an XCOMMIT entry is selected and marked DIRTY, it must be written to the backing store. Note though, that XCOMMIT entries are only needed for incresing performance: When a transaction

aborts, the old value must be restored, i.e. written back to memory - or it can be tagged as XCOMMIT, thus avoiding a write back.

3.2.2.3 Intel TSX Implementations

TSX offer two instruction set extensions to make use of the underlying TM: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). In the Intel® Architecture Instruction Set Extensions Programming Reference [55] they are described as follows:

> HLE is a legacy compatible instruction set extension (comprising the XACQUIRE and XRELEASE prefixes) to specify transactional regions. RTM is a new instruction set interface (comprising the XBEGIN, XEND, and XABORT instructions) for programmers to define transactional regions in a more flexible manner than that possible with HLE. HLE is for programmers who prefer the backward compatibility of the conventional mutual exclusion programming model and would like to run HLE-enabled software on legacy hardware but would also like to take advantage of the new lock elision capabilities on hardware with HLE support. RTM is for programmers who prefer a flexible interface to the transactional execution hardware.

HLE and RTM are well documented and an overview of their API is given in this section. Unfortunately, implementation details of the TM system are not publicly documented, so one can only guess what exactly is done on chip. David Kanter reasons about Haswell's TSX internals in an online article, "Analysis of Haswell's Transactional Memory" [59].

While evaluating TM, the focus has been set on RTM. The XBEGIN and XEND instructions specify the start and end of an RTM code region, XABORT aborts an active transaction. RTM supports transactional nesting (see Section 3.2.1), but only up to an unknown "implementation specific depth of MAX_HLE_NEST_COUNT" [55]. It will always return to the outermost XBEGIN instruction when aborting. In the following, the instructions are described and implementation details are given:

`XBEGIN` - Transaction Begin

Opcode/Instruction	Op/En	64 / 32bit Mode Support	CPUID Feature Flag
C7 F8 XBEGIN rel16	A	V/V	RTM
C7 F8 XBEGIN rel32	A	V/V	RTM

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	Offset	NA	NA	NA

Table 3.3: `XBEGIN` instruction and its operand encoding [55].

Marks the beginning of a transaction and executes it. Provides a 16-bit / 32-bit relative offset to compute the fallback instruction address. The fallback instruction address is stored and can then be used to mark a region in code that the instruction pointer returns to when the active transaction aborts.

`XEND` - Transaction End

Opcode/Instruction	Op/En	64 / 32bit Mode Support	CPUID Feature Flag
0F 01 D5 XEND	A	V/V	RTM

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA

Table 3.4: `XEND` instruction and its operand encoding [55].

Marks the end of a transaction which was set active by a matching `XBEGIN` instruction.

`XABORT` - Transaction Abort

Abort a transaction and continue at the fallback address given by the outermost `XBEGIN` instruction. “The `EAX` register is updated to reflect an `XABORT`

Opcode/Instruction	Op/En	64 / 32bit Mode Support	CPUID Feature Flag
C6 F8 ib XABORT imm8	A	V/V	RTM

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	imm8	NA	NA	NA

Table 3.5: XEND instruction and its operand encoding [55].

instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX" [55].

Based on to these instructions, RTM allows the programmer to implement profound synchronization control structures and different lock elision functionality (which has been done during these studies, see Chapter 4). Solutions based on RTM are best encapsulated into extra classes and libraries, since its syntax and logic is more difficult to use than HLE. It may not be suited to manually use it for each transaction that a programmer wishes to execute.

TSX is a *best effort* hardware implementation. Transactions are limited in size and lack support for several instructions. Executing unsupported instructions within a transaction's context forces an abort. Most notable, I/O and interrupt operations are not supported, since TSX provides no roll-back support for it. The full list can be inspected in [55].

3.2.3 Software Transactional Memory

STM is a design concept of transactional programming that adapts the TM foundations, but not its implementation. The intent of the original proposal by Shavit and Touitou is to provide a shared (software) object "which behaves like a memory that supports multiple changes to its addresses by means of transactions" [81]. This initial protocol only supports static transactions which simplifies error detection, but does not allow dynamic memory interactions. This shortcoming has been overcome by the TL2 STM [26]. Most modern STM designs allow big objects to be handled within transactions and they are not directly limited by hardware capabilities. Most importantly, they provides I/O

support - an important feature that early STM and RTM designs are lacking. Since it is not bound to specific hardware, it comes with better portability and cross platform deployment is easier. STM, too, provides instructions to read and store values of a shared memory location in a transactional region, similar to the `LT` and `ST` instructions outlined in Section 3.2.2.1. Intel's TM ABI [54] is an attempt at providing a common interface in order to access TM and STM implementations in a standardized fashion. GCC provides a STM implementation and makes use of the ABI in form of its `libitm` library. This way, other STM implementations can easily be linked and used instead of the internal version without changing existing code.

From a programmer's perspective, the start and end of a transactional region needs to be specified and all read and write accesses to shared memory locations must be replaced with STM calls. Instead of doing this manually, an STM compiler can replace the memory access calls automatically and the programmer only marks a transactional region in code. In GCC this is done by scoping a code region into transactional blocks:

Listing 3.1: Transactional block in GCC.

```
int shared_data[20];

int
set_shared_data(int index, int value)
{
    __transaction_atomic {
        shared_data[index] = value;
    }
}
```

Relying on a compiler may lead to worse performance compared to manual instrumentation due to *compiler over-instrumentation* [13,30,98], but drastically reduces programming complexities.

STM libraries can be designed in many different ways, following different intentions - and many of them have been implemented over the last two decades. Some aim for low sequential overhead, some focus on being wait-free and others aim for high scalability. This study does not discuss the vast amount of different STM implementation approaches, as Harris et al. discuss this in detail [46]. A small selection of well-known implementations is listed in the following:

- TL2 STM [26], commit-time locking and a global version-clock (a TL2 based STM is available as part of [12])
- TinySTM [33], dynamically chooses its strategy and contains a time-based and lock-based design
- GCC-4.7+, supports STM directly in the compiler
- JVSTM [11], Java implementation based on versioned boxes
- Haskel STM [47], Haskell implementation

For the experimental evaluation TinySTM has been selected, as being identified as a solid and good performing implementation that supports GCC's libitm. It replaced GCC's internal STM for the benchmark setups.

3.3 Developing Software using TM

Listing 3.1 shows how transaction can be specified using the GCC compiler. It appears to be very simple to develop concurrent software using TM. And indeed, in small programs or applications written from scratch, one can easily specify transactions. But also for TM, there are difficulties and problematic situations that should not be ignored. They are outlined in the following paragraphs.

Calling Functions within Transactions

GCC is able to infer transaction safety to a function, if the definition is within the scope of the current source file. If this is not the case, the called function has to be annotated to mark it transaction safe. In existing software that is fairly complex, this may lead to a lot of effort in refactoring code. While trying to make ROOT's `TH1` histogram class thread-safe to get an impression of TM this problem became obvious. Because of its heavily object oriented design, the `Fill()` function calls many other nested functions (for example to internal buffer and axis objects that call functions themselves). Annotating all these functions - that are spread over many different source files - is not enough,

since they really have to be made thread-safe, too. TM is no magic tool that makes an application thread-safe without any effort.

STL and System Calls

When calling external functions like system calls or using STL containers, some functions must be marked transaction safe. `malloc` and `free` are automatically handled by GCC, but in other situations annotations are necessary [68]:

Listing 3.2: STL in transactional code.

```
#include <unordered_map>
std::unordered_map<int, int> ht;

int main() {
    __transaction_atomic {
        ht[1] = 1;
        ht.find(1);
    }
}
```

This code does not compile. GCC outputs that the statement `ht[1] = 1` is an unsafe to be called from within transactions:

Listing 3.3: Compilation Error using unordered map.

```
tm_annotation.cpp:6:13: error: unsafe function call 'std::unordered_map<_Key,
_Tp, _Hash, _Pred, _Alloc>::mapped_type& std::unordered_map<_Key, _Tp, _Hash,
_Pred, _Alloc>::operator[](std::unordered_map<_Key, _Tp, _Hash, _Pred,
_Alloc>::key_type&&) [with _Key = int; _Tp = int; _Hash = std::hash<int>;
_Pred = std::equal_to<int>; _Alloc = std::allocator<std::pair<const int, int> >;
std::unordered_map<_Key, _Tp, _Hash, _Pred, _Alloc>::mapped_type = int;
std::unordered_map<_Key, _Tp, _Hash, _Pred, _Alloc>::key_type = int]'
within atomic transaction
         ht[1] = 1;
             ^
```

It automatically detects that the `find()` method is transaction safe, but the `[]` operator may throw an `std::__throw_bad_alloc` exception which is not defined in this scope. Fixing this issue would require annotating the `unordered_map` class. Currently, there is an ongoing effort to make the STL transaction safe, but it is planned to be shipped with the future GCC version 4.10. The current status and implementation of the transaction safe STL can be monitored at https://github.com/mfs409/tm_stl. At this point of time, the `std::list` container is the only completely transaction safe STL class.

3.4 Criticism and Endorsement

Many researchers consider TM a very promising programming idiom that could overcome drawbacks of the current synchronization mechanisms. But many STM systems are relatively slow and even optimized implementations come with some great overheads. In order to evaluate a TM implementation, Stanford University developed the Stanford Transactional Applications for Multi-Processing (STAMP) [12]. It is a well-known benchmark suite that consists of eight configurable benchmarks.

Many publications [7, 25, 40, 66, 86] present new improved STM algorithms, but only compare them to other STM implementations, leaving the question whether they can compete with established practices or a serial implementation open.

In 2008, Cascaval et al. published their evaluation and future prediction of TM in an article named "Transactional Memory: Why is it Only a Research Toy?" [13]. Their measurements are based on STAMP and show a significant lower performance compared to serial execution. They conclude that lowering STM overheads may be an uphill battle and doubt its feasibility. According to their conclusion, the impact of the TM model itself might be overestimated [13]:

> We observed that the TM programming model itself, whether implemented in hardware or software, introduces complexities that limit the expected productivity gains, thus reducing the current incentive for migration to transactional programming, and the justification at present for anything more than a small amount of hardware support.

More precisely, the specific arguments are:

- STM: comes with overheads that exceed the users' tolerance.
- Hardware TM: has high implementation and verification costs that lead to huge design risks. Current architecture design constraints reduce per-

formance and their error handling adds complexity to the programming model.

- Hybrid TM: is a good concept, but the mix of STM and Hardware TM remains unclear.

Even if overheads would be reduced, the implementations would not fully comply with the idea of easily specifying transactions, but instead break this model through adding complexities such as error handling.

In response to the previous critics, Dragojević et al. extensively analyzed a great number and range of benchmarks and workloads and published their findings in "Why STM can be more than a Research Toy" [29]. Their results differ greatly from the previous findings, since Cascaval et al. provided benchmark settings that enforce high contention workloads and due to hyper-threading throttling performance. While acknowledging that STM is not "a silver bullet for general purpose concurrent programming", it can achieve performance enhancements in many cases and is already usable for some types of applications. In contrast to Cascaval et al., they believe that there is room for improvement and that TM is a promising field of research.

4 Experimental Evaluation

4.1 Experimental Setup

The benchmark system contains an Intel Core i7-4790 CPU, a quad core CPU with eight threads. Each core runs at a frequency of 3.60 GHz and has 32 KB of L1 data cache with a cache line size of 64 bytes. It comes with 16 GB of RAM. The i7 chip belongs to the *Haswell refresh* architecture and therefore natively supports hardware TM. More specifically, it supports TSX (see Chapter 3) with a transaction granularity of 64 bytes. It is important to note that during this study, Intel informed developers about a bug in TSX [56]. This bug can lead to unpredictable system behavior if a complex set of internal timing conditions and system events is met. Therefore, it is advisable to not use TSX in production systems and instead wait for the next chip release. The bug did not occur during this research. All benchmarks have been compiled with `GCC` version 4.9.1 and linked with a custom `glibc` version 2.19 in order to avoid influences of distributor optimizations. All benchmarks have been run with `glibc` configured to support *lock-elision* (`./configure` *options* `–enable-lock-elision=yes`) and `glibc` configured to not support it. No immediate effects on performance behaviour could be observed, however. All code has been compiled using the optimization level `-O3`. In order to use TM with the `GCC` compiler, the `-fgnu-tm`, `-mrtm` and `-mhle` flags have been set.

The next two sections will describe a queue and histogram benchmark studying the effect of synchronization techniques on performance (runtime) and scalability. Both of them have been measured in the same way: All tests are run ten times to warm up the CPU and to fill up the cache lines. Afterwards, they are run 40 times in a row and their runtime is measured by the `StopWatch` class listed

in Appendix A.3. The mean value and standard deviation of this measurement series is calculated. A full benchmark run repeats this procedure eight times. At first, the measuring is done using only a single thread, and then repeated for up to eight threads. While increasing the number of threads, the workload stays constant and it is evenly split between all threads.

4.2 Queue Benchmark

4.2.1 Queue Implementations

Queues are abstract data structures that function as a data collection to which data entities can be added (*enqueued / pushed*) or removed (*dequeued / popped*). A queue enforces a *first-in, first-out* order of data: the oldest entry that has been enqueued is the first to be removed by a dequeue operation. Figure 4.1 illustrates this concept. Queues are widely used data buffers and have been

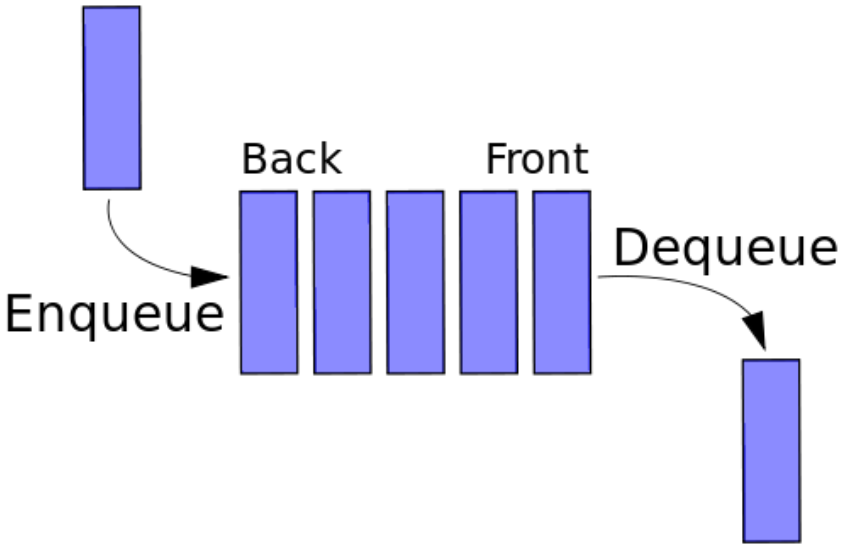

Figure 4.1: Queue data structure [21].

studied for many years. Various implementations exist for fixed size or variable size queues. The following benchmark uses variable size queue implementations with support for concurrency. All class members are padded according to the system's cache line size.

Serial Dynamic Queue

A queue with dynamic memory management. For each element that is pushed into it, a new node is allocated. The dynamic queue is *not* thread-safe and is meant to act as reference implementation for the other concurrent implementations. When measuring speedup, this queue will be the baseline.

Mutex Locked Queue

Extends the serial dynamic queue with *lock guards* - scope based mutexes - to ensure thread-safety.

TM / STM Queue

A transactional queue. Its implementation is based on GCC's TM. The enqueue and dequeue operations are encapsulated into a transactional block (see Listing 4.1 and 4.2).

Listing 4.1: TM Queue - push operation.

```
void
push(const T& element)
{
    Node* tmp = new Node(new T(element));

    __transaction_atomic {
        m_last->next = tmp;
        m_last = tmp;
    }
}
```

Depending on the compiler and linker settings, this queue is making use of the TSX instruction set, i.e. using hardware TM, or it is using an STM backend. For this benchmark, the STM based implementation is linked against TinySTM [33]. Section 3.2.3 explains how GCC's `libitm` is used to achieve this interchangeability. Note that the I/O operation is performed outside of the transaction scope. An STM library would most likely be able to handle the node allocation, but when compiling for TSX usage, the transactions would abort.

Listing 4.2: TM Queue - pop operation.

```
bool
pop(T& result)
{
    Node* first;
    Node* next;

    T* val;

    __transaction_atomic {
        first = m_first;
        next  = first->next;

        // Queue is empty.
        if(next == nullptr) {
            return false;
        }

        val         = next->value;
        next->value = nullptr;
        m_first     = next;
        result      = *val;
    }

    delete val;
    delete first;

    return true;
}
```

Deleting elements is critical as well. A dequeue or pop operation stores the queue's head in a referenced variable and then removes the old values. This has to be kept in mind when implementing transactions: Note how the temporary `Node` pointers are set within the transaction and deleting old `Node` objects is done outside the transaction scope.

RTM Queue

In addition to the TM and STM Queue, another implementation is using RTM directly. For this purpose, an RTM wrapper class has been written to provide synchronization mechanisms based on lock elision. The full implementation can be seen in Listing A.2. The `RTMLock` class provides a locking API that starts an RTM transaction and uses a spinlock based fallback path.

Spinlock Queue

The spinlock implementation is based on a `std::atomic<bool>` variable. During an enqueue operation the value of the atomic boolean flag is tried to be atomically exchanged. If this operation fails, a while loop is used to keep spinning - constantly retrying to change the value of the flag. The exchange is an atomic `read-modify-write` operation: while performing it, the value cannot be affected by any other thread. Listing 4.3 shows the push (enqueue) operation implementation:

Listing 4.3: Spinlock Queue - push operation.

```
void
push(const T& element)
{
    Node* tmp = new Node(new T(element));

    // Acquire exclusivity.
    while(producerLock.exchange(true)) { }

    // Publish to consumers.
    m_last->next = tmp;

    // Swing m_last forward.
    m_last = tmp;

    // Release exclusivity.
    producerLock = false;
}
```

When the variable `producerLock` has successfully been set to `true`, a thread is allowed to enter the critical region. When leaving it, the current thread has to set it back to `false` in order to allow other competing threads to enter the region. The variable name `producerLock` already suggests that the queue is designed in a way such that the *head* and *tail* pointers are protected by two separate locks in order to reduce the critical region.

Lockfree Queue

Lockfree queue based on Michael and Scott's proposal [70], imported through the `boost lockfree` library.

4.2.2 Benchmark Setup

Testing the queues is done through *enqueue* - *dequeue* pairs where each thread performs a certain amount of consecutive *enqueue* - *dequeue* operations. This leads to very high contention. A delay functor object (see Appendix A.4) is used to regulate this contention by executing various levels of workload. Due to the function calls during the workloads, caching behavior becomes an influence and the absolute time it takes to execute is varying, which reduces contention. It provides the following modes of work that can be performed between adding data and removing elements from the queue (with the order of absolute time it takes on the given testing system in brackets):

- **LoadLevel::None:** `no-op` operation, leads to high contention [0ns].
- **LoadLevel::Low:** perform little amount of calculations [270ns].
- **LoadLevel::Medium:** perform calculations to reduce contention [684ns].
- **LoadLevel::High:** perform calculations to nearly eliminate effects of contention [1554ns].

4.2.3 Results

Figure 4.2 shows the measurements for all queue implementations. A bump in the low contention situation curve - the *high* workload situation - when running five threads can be spotted immediately. Remembering the description of the experimental setup in Section 4.1, this can be explained by hyper-threading stepping in. The processor is a quad-core and fully supports four threads, but makes use of Intel's hyper-threading technology when initiating more than four threads. Hyper-threading is a technique that addresses two virtual cores for each physically core present. It distributes workload between those two and shows the operating system two hardware processors. Internally, they still have to share the same hardware resources and they do not properly scale, but hyper-threading increases the throughput of the pipeline. Unfortunately, the benchmark could not have been performed on a system with 32 or more real hardware cores, because of lacking TSX support on these systems.

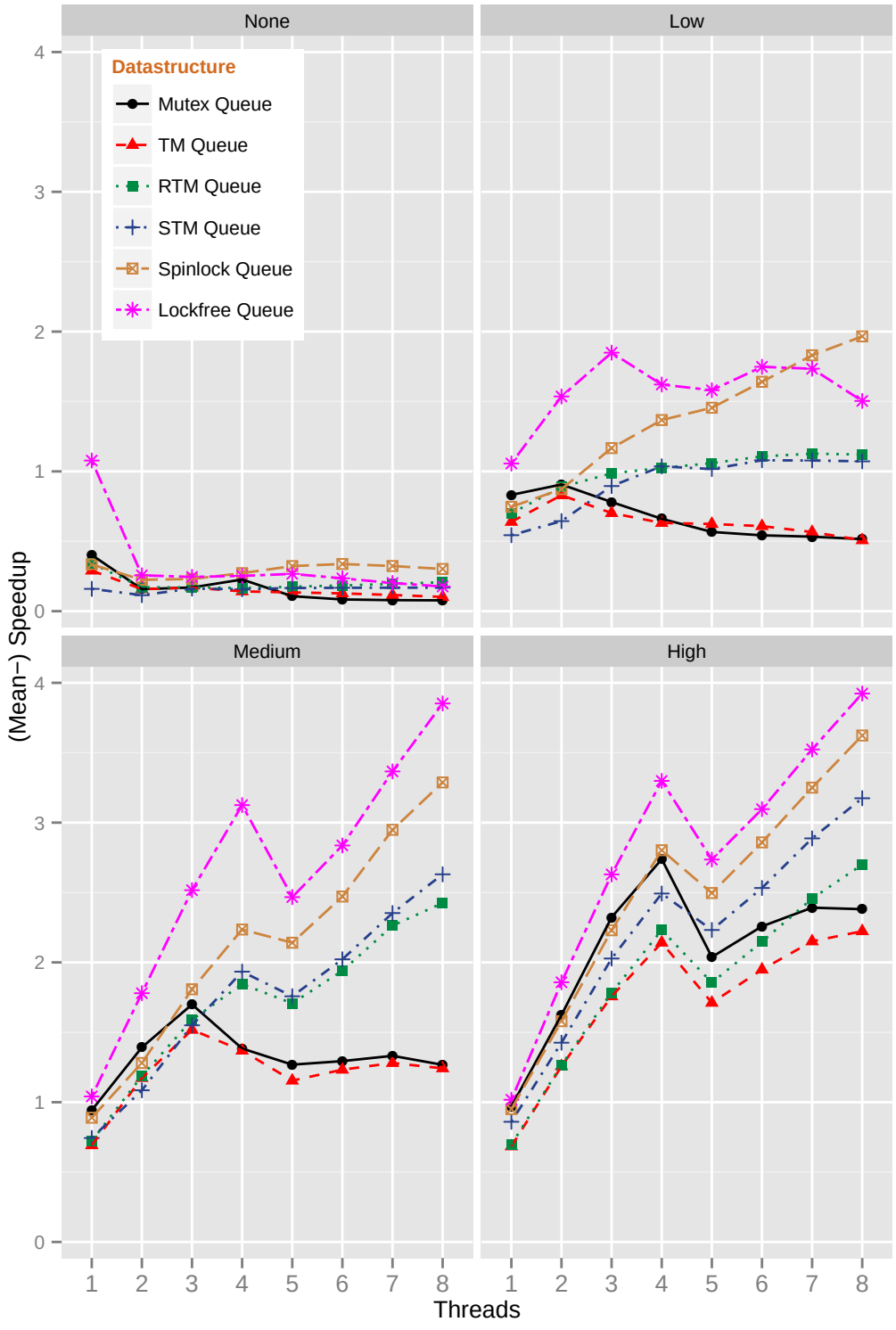

Figure 4.2: Queue Speedup using different techniques.

Besides the bump in the speedup curve, the mutex, lockfree and spinlock based implementations have similar speedup behavior, with the lockfree queue ouperforming the others by a tiny margin. All transactional queues perform worse. Interestingly, the STM based queue performs better than the ones making use of TSX instructions. This may be explained by TinySTM having a specialized contention manager for a certain contention situation. In STM, these contention managers ensure forward progress and balance throughput and fairness between threads. There are different strategies for conflict resolution in transactions. Details can be found in [85].

Extremely high contention (overhead = "None") slows down the application no matter what concurrency control technique is used. The overhead of synchronization leads to a performance decrease in the single threaded case, too, except when using the lockfree queue which completes its operations in the same amount of time (speedup = 1).

The medium workload level speedup curves are very similar for the mutex based and TM based queues. They scale more or less linearly up to a speedup factor of $1.5 - 1.7$ for three threads. After this point the speedup curve flattens out. Likewise, the RTM and STM implementations scale linearly up to a speedup of two for four threads, and then they keep rising up to a speedup factor of 2.5 when hyper-threading kicks in. For the lockfree and spinlock queues, the curves have the same characteristics as their curves in the low contention situation, with the lockfree queue slightly outperforming the spinlock queue. This behavior indicates that these two mechanisms are not as easily affected by contention as their competitors.

When looking at the low workload level curves, it becomes clear that the lockfree queue again outperforms the other queues. The spinlock based queue has a more constant scaling behavior, but reaches the speedup peak of two at a greater number of threads. RTM and STM based implementations settle at a speedup of one when having more than two threads. Before this point they slow down the application. The mutex and TM based queues slowly decrease performance with an increasing number of active threads.

Conclusively, a developer should opt for either a lock-free queue or using spinlocks. With lock-freedom overcoming several drawbacks of mutual exclusion

(see Chapter 2), the lock-free queue would be a good and safe choice. On top of that, it never performed worse than the serial queue. Another observation is that the mutex and TM based queues behave quite similar, except for the peak in the low contention situation. When opting for a transactional implementation, the TinySTM based STM queue is actually outperforming its contenders.

4.3 Histogram Benchmark

4.3.1 Histogram Implementations

A histogram is a graphical representation of the frequency distribution of data over discrete intervals, called *bins*. It serves as a probability distribution estimation and was introduced by Karl Pearson in 1895 [73]. The frequency is accumulated over a fixed amount of bins and plotted as adjoining rectangles:

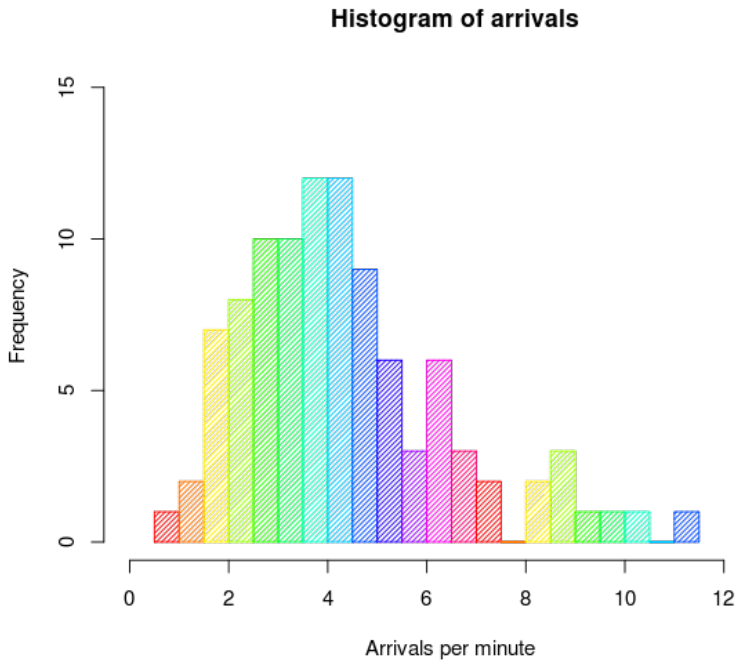

Figure 4.3: Example histogram, that is showing frequencies of arrivals per minute [22].

The area of all rectangles shows the total number of all data. When estimating probability densities, the area of the rectangles has to be normalized to one.

Comparing histograms is a widely used technique in HEP. In doing so, one is able to analyse data by performing statistical hypothesis tests, such as the *chi-squared-test*.

Serial Histogram

Internally, the histogram is using a templated array which acts as a buffer for the histogram, where each array element represents one bin. Values that are added to a histogram may underflow or overflow a predetermined range of the histogram. In order to count them, the array has two additional elements: The very first and last entry are dedicated to count under and overflows. A second array of the same size is used to store the squared sum of each bin, which can be used to determine the *mean squared error* of each bin. Additionally, a counter keeps track of the total number of entries that have been added to the histogram. The histogram uses a fixed number of bins that is specified via its constructor.

Listing 4.4: Histogram - push operation.

```
int
Fill(double x, T weight)
{
    const int bin = FindBin(x);

    fEntries++;

    if (bin < 0) return -1;

    fArray[bin] += weight;
    fSumw2[bin] += weight*weight;

    return bin;
}
```

Locked Histogram

The class `THistogramLock` is based on the serial histogram and protects critical regions through lock guards.

Spinlock Histogram

The spinlock implementation protects the critical region through a spinlock that completely locks it. It is using the `pthread_spinlock_t` type that is provided by the `pthread` library.

TM and RTM Histogram

The hardware TM, STM and RTM implementation all encapsulate the fill operation in a transaction. Ideally, this would lead to very fine-grained locking, where only the conccurent filling of neighboring bins would lead to a transaction abort (two neighboring bins are probably stored in a single cache line).

4.3.2 Benchmark Setup

Benchmarking the histogram implementations is done by simply filling a histogram through its `Fill()` function. The histogram is setup with 1000 bins that represent an interval of $[0, 1000000]$. During the benchmark, a histogram is filled with 1000000 random numbers that are uniformly distributed over this interval. In the sequential benchmark, a single thread performs the filling of a single histogram. In the multi-threaded scenario, a single histogram is filled by N threads. The 1000000 filling operations are then split between the N threads, such that the total workload stays constant for each number of threads. As already described in Section 4.2.2, an artificial workload is induced after each fill operation to control the level of contention.

4.3.3 Results

Figure 4.4 shows the measured speedup for all initial histogram implementations. All implementations have the same speedup characteristics in the high workload scenario for this range of threads, except for the STM histogram reaching a slightly lower speedup. In the medium workload scenario the histograms reach nearly the same speedup as they do in the high workload case, except for the STM and TM histogram, which is lower. The implementations mainly start to differ when being hyper-threaded.

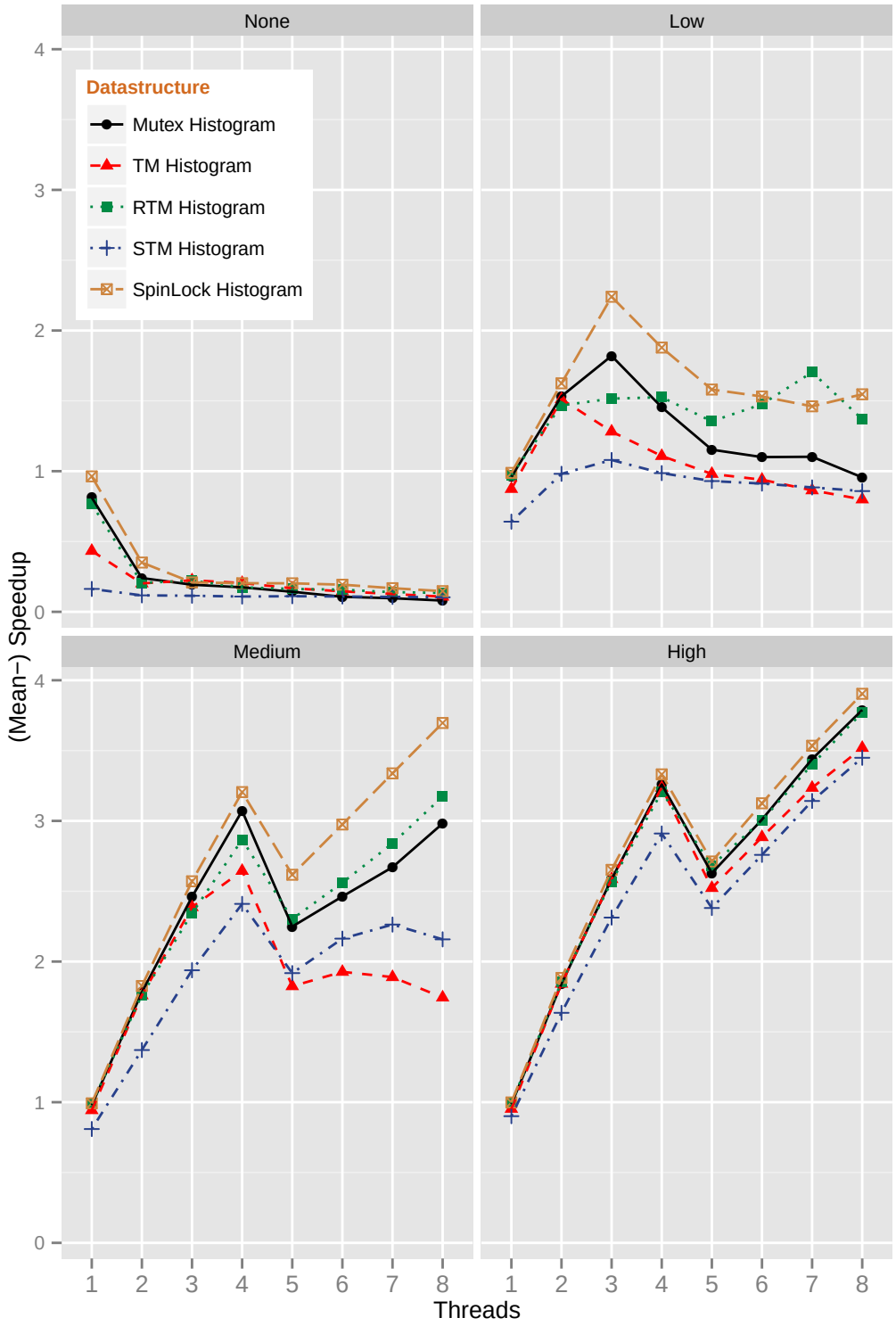

Figure 4.4: Histogram Speedup using different techniques.

In this case, mutexes and RTM perform equally well, but are slightly outperformed by the spinlock implementation. Due to a smaller critical region, both of the two workloads are handled well.

As already seen for the queues, a massive slowdown occurs in the high contention scenario.

When running the histograms in a low workload scenario, an average speedup of 1.5 can be observed, except for the TM and STM histograms, which settle in at a speedup of around one with a trend to slow down with an increasing number of active threads. If one thinks about histograms this may come as a surprise, since this should be a nice example for transaction based synchronization: the bins are implemented as arrays, and accessing them should lead to a memory access pattern that is spread over many cache lines (thus having a low transaction abort rate), especially when filling random numbers that are uniformly distributed. In order to analyze the behavior of transactions, the linux kernel offers TSX specific kernel events that can be analyzed using the `perf` linux profiling utility. Three events are of great interest:

- `tx-start`: event counter for started transactions.
- `tx-abort`: event counter for transaction aborts.
- `tx-capacity`: event counter for transaction aborts due to capacity conflicts.

Running the benchmark through the `perf` profiling tool[6], the diagnostics of Table 4.1 are obtained. They translate into a transaction abort rate of $16,3\%$

TM histogram		RTM histogram	
396,638,978	`tx-start`	639,122,886	`tx-start`
64,963,228	`tx-abort`	251,989,781	`tx-abort`
1,261	`tx-capacity`	71	`tx-capacity`

Table 4.1: TSX transaction diagnostics for the low workload scenario.

[6] Execute the benchmark driver through prepending the following on the command line: `perf stat -e tx-start,tx-abort-tx-capacity`.

and $39,4\%$ - which is quite high. Key to performance oriented TM is keeping the abort rate as low as possible. To do so, the implementation is reworked, such that the central `fEntry` counter is replaced by an array of counters. Figuring that the actual number of total elements is mostly relevant when actually doing analysis with a histogram, the work to add up bin specific counters is negligible. Refactoring this part of the code and aligning member variables to the system's cache line size leads to new transaction diagnostics which are listed in Table 4.2. The total number of transaction aborts is heavily reduced. The new abort ratio is then $5,6\%$ and $8,1\%$.

TM histogram		RTM histogram	
363,677,898	`tx-start`	434,836,615	`tx-start`
20,691,547	`tx-abort`	35,393,855	`tx-abort`
691	`tx-capacity`	141	`tx-capacity`

Table 4.2: TSX transaction diagnostics after refactoring.

Additionally, the mutex based histogram implementation can be refactored to a more fine-grained locking variant. Instead of a lock around the whole critical section, the counter-per-bin method is adopted and a lock for each bin is set up. Note that this increases the memory footprint per histogram object. When designing concurrent applications, developers will always have to balance the trade-off between performance and memory. Figure 4.5 shows the benchmark with the updated and refactored code in place. And indeed, a higher speedup is achieved through refactoring. It is mainly reflected in the low workload case which indicates that a workload level between the medium and high workload makes contention negligible. Fine-grained locking turns out to heavily speed up the mutex based histogram implementation. The RTM based histogram is as good for up to four threads, but it does not scale in hyper-threaded mode. This is due to the fact that TSX has fixed hardware limits concerning the amount of active transactions (L1 cache size). The virtual processors still track the cache line per core, but virtually sharing a processor's resources does not improve the situation.

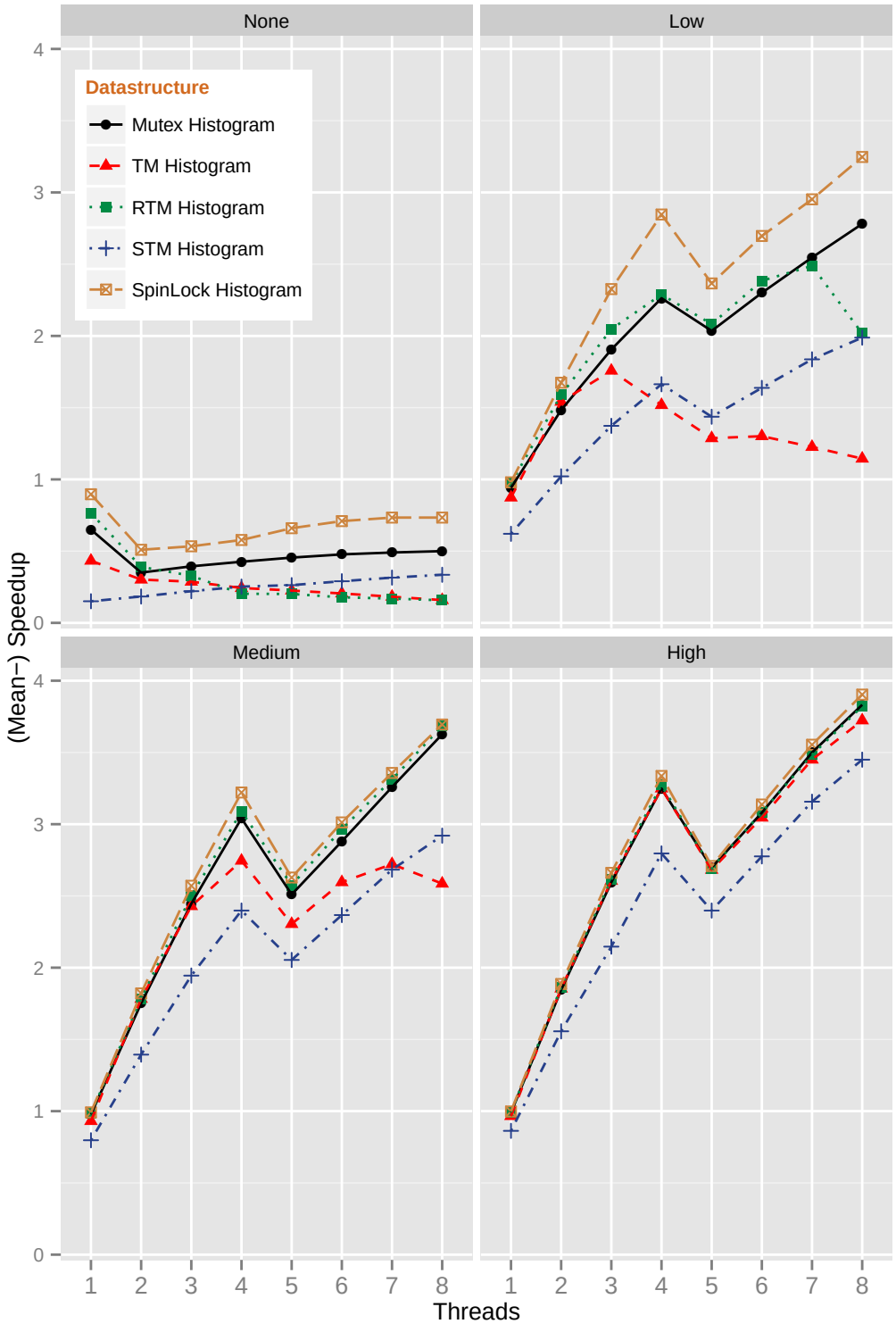

Figure 4.5: Refactored Histogram Speedup using different techniques.

In a further step, Downey's speedup model (see Section 1.4.1) is applied to the refactored histogram classes. Figure 4.6 shows the fitted model plots and lists the average parallelism A and its variance σ for each implementation for all workloads. Since the speedup curves are not monotonically increasing, the high variance model is applied. The low variance model is not able to model these situations and always converges to $A = 1$ for the given measurements. Downey's speedup model can not be applied to the high contention curves, since it does not describe slowdown situations - A has to be within its function domain, that is $A \geq 1$, as shown in Equation 1.5 and Equation 1.6. The TM, RTM and spinlock histograms perform well regarding their variances in parallelism which indicates a more stable program behavior overall. Contrarily, the mutex and STM histograms have a very high variance in parallelism ($\sigma = 3.8664$ and $\sigma = 10$, respectively) for only low workloads.

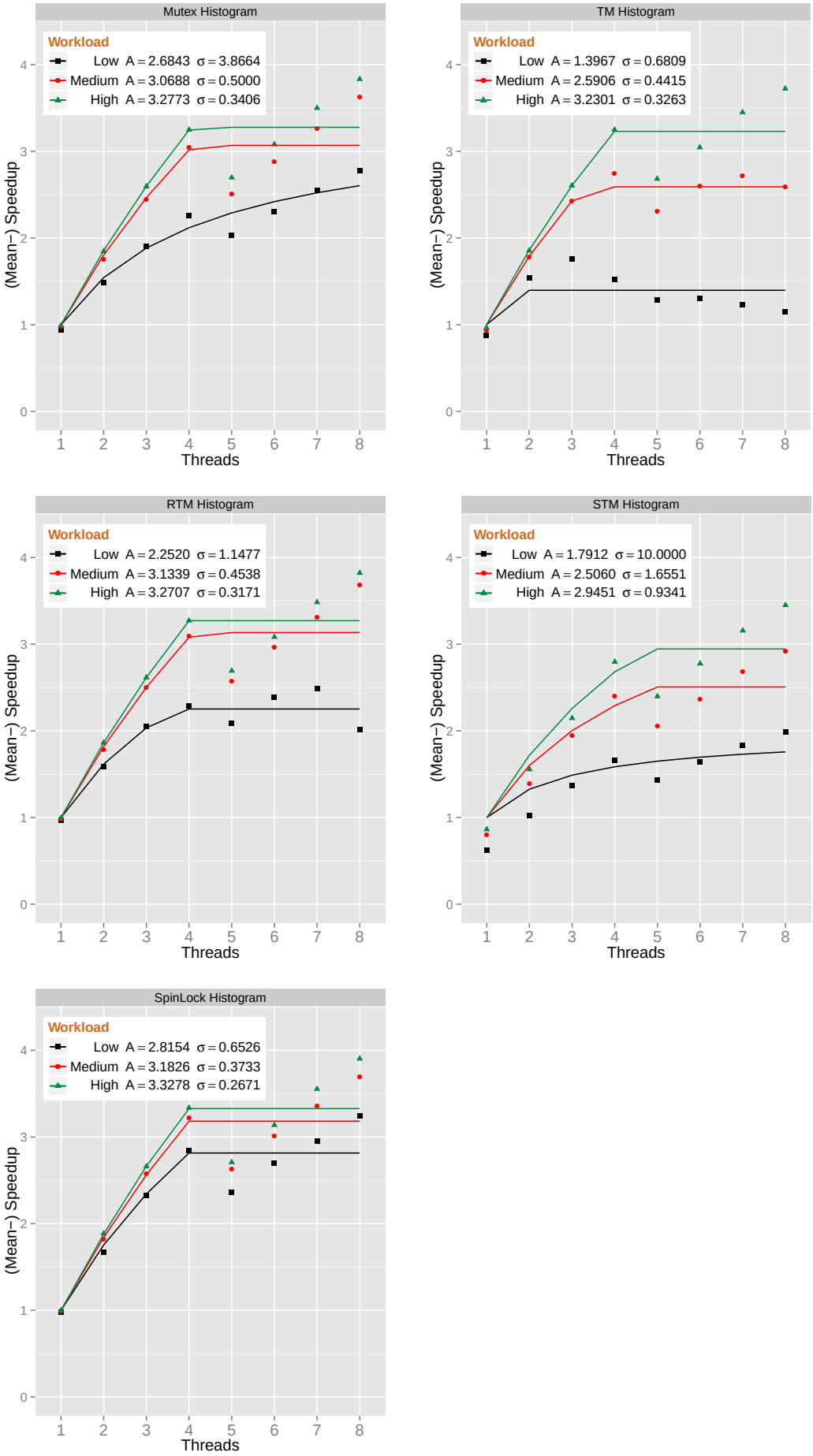

Figure 4.6: Downey Speedup Model applied to refactored Histograms.

4.4 Experimental Evaluation in Literature

Experimental evaluation of TM, especially of hardware TM implementations, is rare and no common conclusion has been drawn with respect to the feasibility and performance of TM. Several benchmark suites exist [12, 40] that allow to investigate and study TM behavior, but most of them do not provide means to easily compare TM to conventional synchronization techniques or sequential implementations. One benchmark suite that does provide such a comparison is Lee-TM [6]. It provides a non-trivial and realistic benchmark, but it only validates STM. In their paper [40], the Lee-TM authors observe that STM is on par with coarse-grained locking, but it never reaches the performance of medium or fine-grained usage of locks. Other research shows that STM implementations can be used successfully, but it is not outperforming conventional locking techniques.

Schindewolf et al. present CLOMP-TM [78], a benchmark that studies hardware TM (embedded into OpenMP) performance on a BlueGene/Q processor prototype system and extract code properties that favor TM. Intel ported this benchmark, STAMP and RMS-TM [60] to be able to make use of TSX. In "Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing" [97] they present their findings and conclusions. In summary, TSX improves existing STM implementations in many cases and in some cases it outperforms even fine-grained locking solutions. However, there are cases where TSX performs worse than STM and the baseline when it is not optimized. With *transactional coarsening* optimizations, they find a significant speedup. Transactional coarsening is a batching technique that is able to statically or dynamically pack several operations into a single transaction. For example, instead of issuing a transaction in each loop iteration, the first iteration could mark a transactions beginning, and only after several other iterations is the transaction end set accordingly.

A similar picture is given by Sylvain Genevès when he ported STAMP applications to RTM instructions, see Figure 4.7. Genevès observes the non-linear scaling due to hyper-threading, too. The *labyrinth* benchmark contained a bug, where it would only use one thread, thus the constant speedup curve. During his analysis he finds that *Yada*, *vacation* and *genome* perform best when using

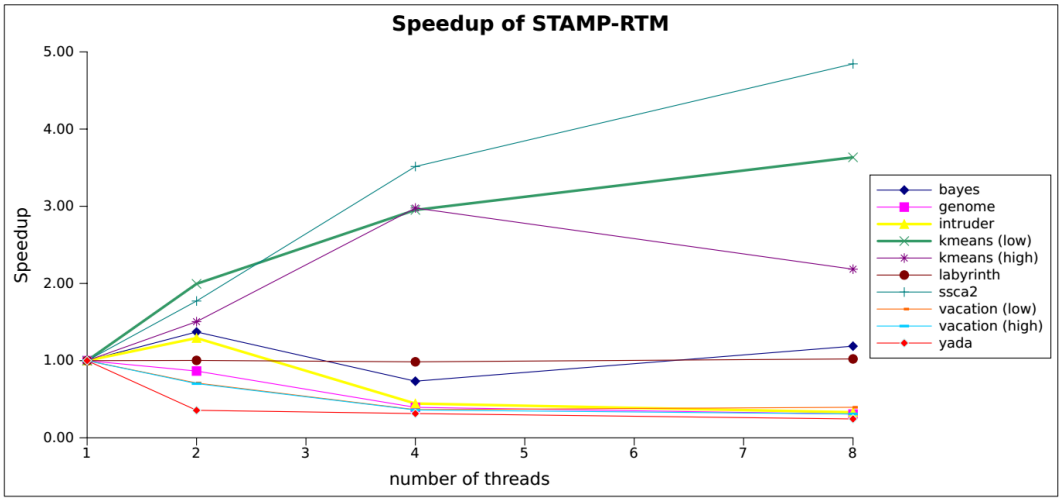

Figure 4.7: Speedup of STAMP-RTM [37].

only one core, while *Intruder* and *Bayes* perform well at two cores with the main issue being the contention in the fallback lock.

Another experimental evaluation focuses on software engineering aspects when using TM: "A Study of Transactional Memory vs. Locks in Practice" [72] compares the approach of graduate-level student programmers to developing concurrent software with locking and TM. It observes that students who make use of locks have more trouble developing correct software, but more effort must be put into the code to make transactions perform well.

5 Conclusions

5.1 Summary

Established concurrency control mechanisms are still the state of the art, but come with drawbacks by design and are expensive to develop and maintain. Message-passing architectures are a good and scalable alternative even in shared memory machines, but as soon as many results have to be synchronized, they become more and more complex. TM aims for simple usage while still providing lock-freedom. During this evaluation study, several aspects of the usage of TM have been demonstrated. The following paragraphs give a summary of the most important conclusions.

Contention is critical

Contention is important when developing parallel applications. High contention almost always stops scaling and decreases performance. This turns out to be even more important when using TM: due to its optimistic design - transactions are executed optimistically and the synchronization mechanism only steps in when an error is detected - TM performs well in low contention scnearios. While it comes with minimal to no overhead in this situation, it performs very poorly in high contention scenarios. Here, the overhead of transaction aborts and roll-backs becomes apparent. Since all up-to-date hardware implementations are bounded, slow path fallback solutions must be provided, which heavily breaches the original intent of lock-freedom and ease of use. It is very advisable to encapsulate this in wrappers that manage hardware transactions, since it is all but easy to use if done manually for every transactional algorithm. STM libraries may adopt to several contention levels and change transaction

scheduling accordingly, but its managing overhead leads to a bad performance in general.

Overall Performance

High contention situations aside, hardware TM is comparable to a coarse-grained usage of mutexes. Out of the box it sometimes performs better, sometimes worse. By analyzing an application's transactional behavior and optimizing it, developers can achieve reasonably good performance. As of today, TM is not optimally suited for the area of HPC, since optimized lock-free and fine-grained usage of locks outperforms TM significantly. To developers who try to exploit multi-processor systems, but who do not need to gain the utmost performance, TM could prove to be the tool of choice. TM may be easier to use for the majority of programmers, but the maximum speedup that can be accomplished seems to be greater when designing optimized data structures relying on atomic operations and locks. Similar conclusions are drawn in [60, 78, 97]. Studying TM and locks in practice [72], Pankratius and Adl-Tabatabai observe that a combination of the two may be a realistic and useful approach.

Ease of use

One major goal of TM is to provide an easy to use mechanism to design concurrent algorithms. This seems to be true for simple classes and programs, but when having complex hierarchies and dependencies, it becomes more and more complex to define transactions. A lot of annotations have to be added to functions and as of now, major tools like the STL cannot be used. Making existing complex software thread-safe using TM turned out to be very burdensome, due to this. On the other hand, building a transactional application straight from the ground may prove viable. Brett Hall showed in his talk at CppCon 2014 that companies can ship software that is built around STM [44]. Things may be further improved with the introduction of the transactional memory language extension to the C++ standard, and with new generations of TSX. Pankratius and Adl-Tabatabai [72] note that teams working with locks spent more time on debugging, whereas students using TM spent more time on achieving good performance. Further research into more optimized or unbounded hardware might reduce the development effort that is needed to gain good performance. TM does provide one important feature

that locking techniques lack, which is the ability to write more generic code through templates (example taken from Michael Wong [94]):

Listing 5.1: A generic template function.

```
template <class T>
void f(T &x, T y) {
    unique_lock<mutex> _(m2);

    x = y;
}
```

When invoking the = operator, no assumptions about the implementation of `T` can be made. It could try to acquire a lock which is also used by other functions and would require a certain order of locking invocations. Essentially, locking and generic programming do not collude very well, whereas TM excels in this use-case due to its composability.

5.2 Other Use Cases for TM

Besides the classic scenario of designing concurrent software for multiprocessor systems, there are ideas for other use cases where TM might prove useful.

Developing software on GPGPU following the SIMD approach can sometimes also require the use of atomic operations or custom-made locks. On GPUs, threads can communicate through an intra-core memory, but blocks must access a global shared memory. Inter-block communication could be synchronized by such custom locks. Almost naturally, this leads to the investigation of TM on graphic processors [14, 36, 95].

Christof Fetzer and Pascal Felber describe how TM can be useful for dependable embedded systems [34]: it opens up another approach to failure control and it may prove useful for real-time systems. Each transaction must be rolled back in the case of an error. In 1975 Randell describes *recovery blocks* [75]. Code may be executed in blocks, and the current state is then checked in a post-condition. If defects are detected, an alternative code is executed. This may be an interesting use-case that could also be modeled with TM. FaulTM is a an STM and hardware TM hybrid that is able to deliver this fault tolerance [96]. Due to their

near to wait-free properties, TM could be adapted to real-time systems. Current TM designs may only be sufficient for soft real-time (see [82]) requirements. Proposals for TM with real-time support are presented in [76, 80].

In "STM Versus Lock-based Systems: An Energy Consumption Perspective" [61], Klein et al. performed a study on energy consumption of STM implementations and compared it to locks. Although they find an energy-inefficiency of a factor of around three on average for current implementations, they believe that these deficiencies could possibly be tackled, reducing energy consumption. With TSX being on the market, power consumption measurements comparing TM and locks could be an interesting research field for the future. Energy consumption is an increasing concern in the fields of grid computing and embedded systems.

TM might also be useful in the field of security, where a failing transaction would translate into a malicious memory access. For this purpose, the managed cache capacity needs to be improved for hardware TM.

5.3 Outlook

As of today, TM is still in an emerging phase towards higher efficiency and mainstream acceptance. New or reworked hardware implementations will need to stabilize and improve performance with coming generations. TSX is a big step forward, and future generations could make TM an appealing approach to a wide range of developers. Cleverly designed hybrid TM libraries could help to reduce I/O related aborts, while still performing reasonably well. In the near future however, TM will most likely not reach the performance of optimized lock-free or fine-grained lock implementations. With C++ getting a transactional memory language extension in the near future, TM popularity could increase to a point wher it becomes feasible for hardware manufacturers to investigate unbound hardware TM. The ongoing effort to make the STL transaction safe is an important step towards a feasible TM mechanism in C++.

Bibliography

[1] Advanced Micro Devices Inc, *Advanced Synchronization Facility - Proposed Architectural Specification*, Mar. 2009. [Online]. Available: http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf

[2] Y. Afek, U. Drepper, P. Felber, C. Fetzer, V. Gramoli, M. Hohmuth, E. Riviere, P. Stenström, O. S. Unsal, W. Maldonado, D. Harmanci, P. Marlier, S. Diestelhorst, M. Pohlack, A. Cristal, I. Hur, A. Dragojevic, R. Guerraoui, M. Kapalka, S. Tomic, G. Korland, N. Shavit, M. Nowack, and T. Riegel, "The Velox Transactional Memory Stack." *IEEE Micro*, vol. 30, no. 5, pp. 76–87, 2010.

[3] S. Agostinelli *et al.*, "Geant4—a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250 – 303, 2003.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 316–327.

[5] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, Jan. 1990.

[6] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis, "Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory," in *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, ser. ICA3PP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 196–207.

[7] M. Ansari, C. Kotselidis, I. Watson, C. C. Kirkham, M. Luján, and K. Jarvis, "Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory." in *ICA3PP*, ser. Lecture Notes in Computer Science, A. G. Bourgeois and S.-Q. Zheng, Eds., vol. 5022. Springer, 2008, pp. 196–207.

[8] I. T. Association, "About InfiniBand." [Online]. Available: http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband

[9] B. Barney, "Using the Sequoia and Vulcan BG/Q Systems," 2014. [Online]. Available: https://computing.llnl.gov/tutorials/bgq/index.html

[10] R. Brun and F. Rademakers, "ROOT — An object oriented data analysis framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1–2, pp. 81 – 86, 1997, new Computing Techniques in Physics Research V.

[11] J. Cachopo and A. Rito-Silva, "Versioned Boxes As the Basis for Memory Transactions," *Sci. Comput. Program.*, vol. 63, no. 2, pp. 172–185, Dec. 2006.

[12] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[13] 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*, vol. 6, no. 5, pp. 40:46–40:58, Sep. 2008.

[14] D. Cederman, P. Tsigas, and M. T. Chaudhry, "Towards a Software Transactional Memory for Graphics Processors," in *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, ser. EG PGV'10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 121–129.

[15] CERN, "The accelerator complex," Sep. 2014. [Online]. Available: http://home.web.cern.ch/about/accelerators

[16] A. Chang and M. Mergen, "801 Storage: Architecture and Programming," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 109–110, Nov. 1987.

[17] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A High-Performance Sparc CMT Processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, 2009.

[18] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon, "Use of Application Characteristics and Limited Preemption for Run-to-completion Parallel Processor Scheduling Policies," *SIGMETRICS Perform. Eval. Rev.*, vol. 22, no. 1, pp. 33–44, May 1994.

[19] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, "Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack," in *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. New York, NY, USA: ACM, 2010, pp. 27–40.

[20] S. Cittolin, A. Rácz, and P. Sphicas, *CMS The TriDAS Project: Technical Design Report, Volume 2: Data Acquisition and High-Level Trigger. CMS trigger and data-acquisition project*, ser. Technical Design Report CMS. Geneva: CERN, 2002.

[21] W. Commons. (2009, Aug.) Data Queue. [Online]. Available: https://commons.wikimedia.org/wiki/File:Data_Queue.svg

[22] W. Commons. (2010, Feb.) Histogram of Arrivals per Minute. [Online]. Available: https://commons.wikimedia.org/wiki/File:Histogram_of_arrivals_per_minute.svg

[23] W. Commons. (2011, May) Cern Accelerator Complex. [Online]. Available: https://commons.wikimedia.org/wiki/File:Cern-accelerator-complex.svg

[24] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with Readers and Writers," *Commun. ACM*, vol. 14, no. 10, pp. 667–668, Oct. 1971.

[25] D. Dice and N. Shavit, "What Really Makes Transactions Faster?" in *Proc. of the 1st TRANSACT 2006 workshop*, 2006, electronic, no. page numbers.

[26] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proceedings of the 20th International Conference on Distributed Computing*, ser. DISC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 194–208.

[27] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Commun. ACM*, vol. 8, no. 9, pp. 569–, Sep. 1965.

[28] A. B. Downey, "A Model For Speedup of Parallel Programs," Berkeley, CA, USA, Tech. Rep., 1997. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1997/CSD-97-933.pdf

[29] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui, "Why STM Can Be More Than a Research Toy," *Commun. ACM*, vol. 54, no. 4, pp. 70–77, Apr. 2011.

[30] A. Dragojević, Y. Ni, and A.-R. Adl-Tabatabai, "Optimizing Transactions for Captured Memory," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 214–222.

[31] U. Drepper, "What Every Programmer Should Know About Memory," Nov. 2007. [Online]. Available: http://www.akkadia.org/drepper/cpumemory.pdf

[32] 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*, vol. 19, no. 11, pp. 624–633, Nov. 1976.

[33] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory." in *PPOPP*, S. Chatterjee and M. L. Scott, Eds. ACM, 2008, pp. 237–246.

[34] C. Fetzer and P. Felber, "Transactional Memory for Dependable Embedded Systems," in *7th Workshop on Hot Topics in System Dependability (HotDep'11)*, June 2011.

[35] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 3.0," Sep. 2012. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[36] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 296–307.

[37] S. Genvès, "Work Report: Lessons learned on RTM," Sep. 2013. [Online]. Available: http://sgeneves.wdfiles.com/local--files/talks/RTM.pdf

[38] J. R. Goodman, "Using Cache Memory to Reduce Processor-memory Traffic," *SIGARCH Comput. Archit. News*, vol. 11, no. 3, pp. 124–131, Jun. 1983.

[39] L. Groves, "Verifying Michael and Scott's Lock-free Queue Algorithm Using Trace Reduction," in *Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory - Volume 77*, ser. CATS '08. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2008, pp. 133–142.

[40] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: A Benchmark for Software Transactional Memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 315–324, Mar. 2007.

[41] J. Guo, "Shared Memory," Aug. 2013. [Online]. Available: http://www.cs.uiuc.edu/class/sp06/cs523/lectures/15/SharedMemory.pdf

[42] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988.

[43] T. Haerder and A. Reuter, "Principles of Transaction-oriented Database Recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983.

[44] B. Hall, "Software Transactional Memory, For Reals." [Online]. Available: https://github.com/CppCon/CppCon2014/blob/master/Lightning%20Talks/Software%20Transactional%20Memory/Software%

20Transactional%20Memory%2C%20For%20Reals%20-%20Brett%20Hall.pdf

[45] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar. 2012.

[46] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory, 2nd edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[47] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable Memory Transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 48–60.

[48] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[49] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 5, pp. 745–770, Nov. 1993.

[50] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-free Data Structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993.

[51] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[52] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[53] iMatix, "Multithreaded Magic with 0MQ," 2010. [Online]. Available: http://www.zeromq.org/whitepapers:multithreading-magic

[54] Intel Corporation, "Intel Transactional Memory Compiler and Runtime Application Binary Interface," Nov. 2008. [Online]. Available: http://software.intel.com/sites/default/files/m/5/a/2/a/f/8097-Intel_TM_ABI_1_0_1.pdf

[55] Intel Corporation, "Intel Architecture Instruction Set Extensions Programming Reference," Intel, Tech. Rep. 319433-014, August 2012. [Online]. Available: http://download-software.intel.com/sites/default/files/319433-014.pdf

[56] Intel Corporation, "Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family - Specification Update," Jun. 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf

[57] Intel Corporation, "Intel IA-64 Architecture Software Developer's Manual: Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C," pp. 353–360, Jun. 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[58] C. Jacobi, T. Slegel, and D. Greiner, "Transactional Memory Architecture and Implementation for IBM System Z," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 25–36.

[59] D. Kanter, "Analysis of Haswell's Transactional Memory," Feb. 2012. [Online]. Available: http://www.realworldtech.com/haswell-tm/

[60] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero, "RMS-TM: a comprehensive benchmark suite for transactional memory systems," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 335–346, Sep. 2011.

[61] F. Klein, A. Baldassin, J. Moreira, P. Centoducatte, S. Rigo, and R. Azevedo, "STM Versus Lock-based Systems: An Energy Consumption Perspective," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '10. New York, NY, USA: ACM, 2010, pp. 431–436.

[62] T. Knight, "An Architecture for Mostly Functional Languages," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ser. LFP '86. New York, NY, USA: ACM, 1986, pp. 105–112.

[63] A. Kogan and E. Petrank, "Wait-free Queues with Multiple Enqueuers and Dequeuers," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 223–234.

[64] A. Kogan and E. Petrank, "A Methodology for Creating Fast Wait-free Data Structures," *SIGPLAN Not.*, vol. 47, no. 8, pp. 141–150, Feb. 2012.

[65] C. Lefevre, "CERN brochure (English version)," Brochure, 2010. [Online]. Available: http://cds.cern.ch/record/1278456/files/CERN-Brochure-2010-005-Eng.pdf?version=1

[66] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, "Anatomy of a scalable software transactional memory," in *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'09*, 2009.

[67] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management." in *PPOPP*, R. Govindarajan, D. A. Padua, and M. W. Hall, Eds. ACM, 2010, pp. 79–90.

[68] P. Marlier. (2012, May) Brief Transactional Memory GCC Tutorial. [Online]. Available: http://pmarlier.free.fr/gcc-tm-tut.html

[69] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.

[70] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96. New York, NY, USA: ACM, 1996, pp. 267–275.

[71] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April 1965.

[72] V. Pankratius and A.-R. Adl-Tabatabai, "A Study of Transactional Memory vs. Locks in Practice," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 43–52.

[73] K. Pearson, "Contributions to the Mathematical Theory of Evolution. II. Skew Variation in Homogeneous Material," *Philosophical Transactions. Royal Society of London. Series A. Mathematical and Physical Sciences.*, vol. 186, pp. 343–414, 1895.

[74] D. H. Perkins, *Introduction to High Energy Physics*, 4th ed. Cambridge University Press, 2000, cambridge Books Online.

[75] B. Randell, "System Structure for Software Fault Tolerance," *SIGPLAN Not.*, vol. 10, no. 6, pp. 437–449, Apr. 1975.

[76] T. Sarni, A. Queudet, and P. Valduriez, "Real-Time Support for Software Transactional Memory." in *RTCSA*. IEEE Computer Society, 2009, pp. 477–485.

[77] W. H. Scharf and F. T. Cole, *Particle accelerators and their uses; 3rd ed.*, ser. Accel. Storage Rings. Chur: Harwood, 1986, enlarged and updated version of the Polish ed.

[78] M. Schindewolf, B. Bihari, J. Gyllenhaal, M. Schulz, A. Wang, and W. Karl, "What Scientific Applications Can Benefit from Hardware Transactional Memory?" in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 90:1–90:11.

[79] M. Schindewolf, A. Cohen, W. Karl, A. Marongiu, and L. Benini, "Towards Transactional Memory Support for GCC," in *GROW '09: GCC Research Opportunities Workshop*, Jan. 2009, affiliated to HiPEAC'09. [Online]. Available: http://gcc.gnu.org/wiki/GROW-2009?action=AttachFile&do=view&target=03-Transactions-Schwindewolf.pdf

[80] M. Schoeberl, F. Brandner, and J. Vitek, "RTTM: Real-time Transactional Memory," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 326–333.

[81] N. Shavit and D. Touitou, "Software Transactional Memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. New York, NY, USA: ACM, 1995, pp. 204–213.

[82] K. G. Shin and P. Ramanathan, "RealTime Computing: A New Discipline of Computer Science and Engineering," in *Proceedings of IEEE, Special Issue on Real-Time Systems*, 1994.

[83] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

[84] V. Smiljkovic, M. Nowack, N. Miletic, T. Harris, O. S. Ünsal, A. Cristal, and M. Valero, "TM-dietlibc: A TM-aware Real-World System Library." in *IPDPS*. IEEE Computer Society, 2013, pp. 1266–1274.

[85] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "A Comprehensive Strategy for Contention Management in Software Transactional Memory," *SIGPLAN Not.*, vol. 44, no. 4, pp. 141–150, Feb. 2009.

[86] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott, "Nonblocking Transactions Without Indirection Using Alert-on-update," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07. New York, NY, USA: ACM, 2007, pp. 210–220.

[87] J. Sreeram, R. Cledat, T. Kumar, and S. Pande, "RSTM : A Relaxed Consistency Software Transactional Memory for Multicores." in *PACT*.

IEEE Computer Society, 2007, p. 428.

[88] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[89] H. Sutter, "Atomic Weapons: The C++ Memory Model and Modern Hardware," 2012. [Online]. Available: http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

[90] H. Sutter. (2013, Aug.) Eliminate False Sharing. [Online]. Available: http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206

[91] Wikipedia, "False sharing - Wikipedia, The Free Encyclopedia," 2014. [Online]. Available: http://en.wikipedia.org/w/index.php?title=False_sharing&oldid=610065846

[92] A. Williams, *C++ Concurrency in Action: Practical Multithreading*, ser. Manning Pubs Co Series. Manning, 2012.

[93] M. Wong, V. Luchangco, J. Maurer, M. Moir, and et al., "Transactional Memory Support for C++," Jan. 2014. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3859.pdf

[94] M. Wong, "Transactional Language Constructs for C++." [Online]. Available: https://github.com/CppCon/CppCon2014/blob/master/Presentations/What%20did%20C%2B%2B%20do%20for%20Transactional%20Memory/What%20did%20C%2B%2B%20do%20for%20Transactional%20Memory%20-%20Michael%20Wong%20-%20CppCon%202014.pdf

[95] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian, "Software Transactional Memory for GPU Architectures," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:10.

[96] G. Yalcin, O. Unsal, I. Hur, A. Cristal, and M. Valero, "FaulTM: Fault-Tolerance Using Hardware Transactional Memory," in *Pespma 2010*

- *Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*, Wei Liu and Scott Mahlke and Tin-fook Ngai, Ed., Saint Malo, France, Jun. 2010. [Online]. Available: https://hal.inria.fr/inria-00494285

[97] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance Evaluation of Intel&Reg; Transactional Synchronization Extensions for High-performance Computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 19:1–19:11.

[98] 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 *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08. New York, NY, USA: ACM, 2008, pp. 265–274.

[99] T. L. Yu, "CSE Operating Systems Concepts and Theory - Lecture Notes," Mar. 2010. [Online]. Available: http://cse.csusb.edu/tongyu/courses/cs660/notes/deadlock.php

Appendices

A Source Code

Lock-free Queue Implementation

Listing A.1: Lock-free Queue Implementation.

```
structure pointer_t { ptr: pointer to node_t,
                      count : unsigned integer }

structure node_t    { value: data type, next : pointer_t }

structure queue_t   { Head: pointer_t, Tail: pointer_t }



initialize(Q: pointer to queue_t)
    # Allocate a free node
    node = new_node()

    # Make it the only node in the linked list
    node->next.ptr = NULL

    # Both Head and Tail point to it
    Q->Head = Q->Tail = node



enqueue(Q: pointer to queue_t, value: data type)
    # Allocate a new node from the free list
    node = new_node()

    # Copy enqueued value into node
    node->value = value

    # Set next pointer of node to NULL
    node->next.ptr = NULL

    # Keep trying until Enqueue is done
    loop
        #Read Tail.ptr and Tail.count together
        tail = Q->Tail
```

```
        #Read next ptr and count fields together
        next = tail.ptr->next

        # Are tail and next consistent?
        if tail == Q->Tail
            # Was Tail pointing to the last node?
            if next.ptr == NULL
                # Try to link node at the end of the linked list
                if CAS(&tail.ptr->next,next,<node,next.count+1>)
                    # Enqueue is done. Exit loop
                    break
                endif
            else
                # Try to swing Tail to the next node
                CAS(&Q->Tail,tail,<next.ptr,tail.count+1>)
            endif
        endif
    endloop

    # Enqueue is done. Try to swing Tail to the inserted node
    CAS(&Q->Tail,tail,<node,tail.count+1>)



dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
    loop
        head = Q->Head
        tail = Q->Tail
        next = head->next
        if head == Q->Head
            if head.ptr == tail.ptr
                if next.ptr == NULL
                    return FALSE
                endif
                CAS(&Q->Tail, tail,<next.ptr,tail.count+1>
            else
                *pvalue = next.ptr->value
                if CAS(&Q->Head,head,<next.ptr,head.count+1>)
                    break
                endif
            endif
        endif
    endloop
    free(head.ptr)
    return TRUE
```

RTM Wrapper Class

Listing A.2: RTM Wrapper Class.

```
#pragma once

#include <xmmintrin.h>

#include "util/Backoff.h"

constexpr unsigned kBackoffSteps = 10;

#define _ABORT_LOCK_BUSY    0xff

class RTMLock
{
public:

    RTMLock()
    {
        pthread_spin_init(&m_lock, PTHREAD_PROCESS_PRIVATE);
    }

    ~RTMLock() { }

    void
    Lock()
    {
        unsigned status;

        for (unsigned retry = 0; retry < kBackoffSteps; ++retry) {
            if ((status = _xbegin()) == _XBEGIN_STARTED) {
                // Hacky check whether spinlock is locked.
                if ((int)m_lock != 1) {
                    // Lock was busy - abort.
                    _xabort(_ABORT_LOCK_BUSY);
                }

                return;
            }


            // From Intel TSX Recommendations:
            //
            // Software can use the abort information provided in
            // the EAX register to develop heuristics as to when
            // to retry the transactional execution and when to
            // fallback and explicitly acquire the lock. For
            // example, if the _XABORT_RETRY bit is clear, then
            // retrying the transactional execution is likely to
            // result in another abort. The fallback handler
            // should distinguish this situation from cases where
```

```
            // the lock was not free (for example, the
            // _XABORT_EXPLICIT bit is set but the _XABORT_CODE()
            // returns a 0xff identifying the condition as a
            // "lock busy" condition). In those cases, the fallback
            // handler should eventually retry after waiting.

            if (!(status & _XABORT_RETRY)) {

                if ((status & _XABORT_EXPLICIT)
                    && _XABORT_CODE(status) == _ABORT_LOCK_BUSY) {
                    for (int i = 0; i < 3; ++i) {
                        if (pthread_spin_trylock(&m_lock) == 0) {
                            return;
                        }
                        _mm_pause();
                    }
                }

                break;
            }
        }

        // Standard lock, since transaction failed.
        pthread_spin_lock(&m_lock);
    }

    void
    Unlock()
    {
        // Hacky check whether spinlock is locked.
        if ((int)m_lock != 1) {
            pthread_spin_unlock(&m_lock);
        } else {
            _xend();
        }
    }

private:

    pthread_spinlock_t m_lock;
};
```

StopWatch Class

Listing A.3: StopWatch Class.

```
#pragma once
#include <chrono>

using Clock = std::chrono::high_resolution_clock;

template<typename Clock = std::chrono::high_resolution_clock,
         typename Rep = std::chrono::nanoseconds>
class Stopwatch
{
public:
    Stopwatch() : m_time_point(Clock::now()) { }

    void
    Reset()
    {
        m_time_point = Clock::now();
    }

    Rep
    Elapsed() const
    {
        return std::chrono::duration_cast<Rep>(Clock::now() -
                m_time_point);
    }

    std::chrono::nanoseconds
    ElapsedNS() const
    {
        return std::chrono::duration_cast<std::chrono::nanoseconds>
                (Clock::now() - m_time_point);
    }

    std::chrono::milliseconds
    ElapsedMS() const
    {
        return std::chrono::duration_cast<std::chrono::milliseconds>
                (Clock::now() - m_time_point);
    }

    std::chrono::seconds
    ElapsedS() const
    {
        return std::chrono::duration_cast<std::chrono::seconds>
            (Clock::now() - m_time_point);
    }
private:
    typename Clock::time_point m_time_point;
};
```

Delay Functor Class

Listing A.4: DelayFunctor Class.

```
class DelayFunctor
{
public:
    DelayFunctor() { }

    void operator() (LoadLevel level, double seed) {
        switch(level) {
        case LoadLevel::LVL_NONE:
            break;
        case LoadLevel::LVL_SLEEP:
            std::this_thread::sleep_for(std::chrono::nanoseconds(5));
            break;
        case LoadLevel::LVL_LOW:
            calc2(seed);
            calc3(seed);
            calc1(seed);
            calc3(seed);
            calc1(seed);
            calc2(seed);
            break;
        case LoadLevel::LVL_MEDIUM:
            for (int i = 0; i < 5; ++i) {
                calc2(seed);
                calc3(seed);
                calc1(seed);
                calc3(seed);
                calc2(seed);
                calc1(seed);
            }
            break;
        case LoadLevel::LVL_HIGH:
            for (int i = 0; i < 10; ++i) {
                calc2(seed);
                calc2(seed);
                calc3(seed);
                calc1(seed);
                calc2(seed);
                calc3(seed);
                calc3(seed);
                calc1(seed);
                calc1(seed);
            }
            break;
        }
    }

private:
    void calc1(double seed) {
```

```
        double x = atan(seed) * 12.5;

        x += 3.14 * seed * 7 * 23 - 1000;

        double z = std::pow(x,2);

        z += sin(seed*1.02);
    }

    void calc2(double seed) {
        double x = atanh(seed) * 12.5;

        x += cos(seed) + 3.14 * seed * 7 * 23 - 1000;

        double z = std::pow(x,2);
        ++z;

        double t = sqrt(10*seed);

        t += std::pow(2,t);
    }

    void calc3(double seed) {
        double x = 1.024 * seed;
        double z = std::pow(x,2);

        double t = sqrt(10*seed);

        t += std::pow(2,t);

        z *= seed * (seed / cos(2.3));
    }
};
```

B Queue Measurements

SerialDynamicQueue

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Serial Queue	1000000	1	41	0	None
Serial Queue	1000000	2	41	0	None
Serial Queue	1000000	3	41	0	None
Serial Queue	1000000	4	41	0	None
Serial Queue	1000000	5	41	0	None
Serial Queue	1000000	6	41	0	None
Serial Queue	1000000	7	41	0	None
Serial Queue	1000000	8	41	0	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Serial Queue	1000000	1	262	0	Low
Serial Queue	1000000	2	262	0	Low
Serial Queue	1000000	3	262	0	Low
Serial Queue	1000000	4	262	0	Low
Serial Queue	1000000	5	262	0	Low
Serial Queue	1000000	6	262	0	Low
Serial Queue	1000000	7	262	0	Low
Serial Queue	1000000	8	262	0	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Serial Queue	1000000	1	625	0	Medium
Serial Queue	1000000	2	625	0	Medium
Serial Queue	1000000	3	630.375	17.7294	Medium
Serial Queue	1000000	4	625.125	0.800641	Medium
Serial Queue	1000000	5	625.3	0.5547	Medium
Serial Queue	1000000	6	625.225	0.531085	Medium
Serial Queue	1000000	7	655.6	29.8397	Medium
Serial Queue	1000000	8	662.55	85.7617	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Serial Queue	1000000	1	1392	0	High
Serial Queue	1000000	2	1392	0	High
Serial Queue	1000000	3	1392	0	High
Serial Queue	1000000	4	1392	0	High
Serial Queue	1000000	5	1392	0	High
Serial Queue	1000000	6	1392	0	High
Serial Queue	1000000	7	1391.97	1	High
Serial Queue	1000000	8	1391.97	1	High

RTMQueue

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Queue	1000000	1	120	0	None
RTM Queue	1000000	2	238.975	1.88788	None
RTM Queue	1000000	3	241.675	1.77591	None
RTM Queue	1000000	4	247.05	1.35873	None
RTM Queue	1000000	5	234.575	1.1209	None
RTM Queue	1000000	6	222.85	1.08604	None
RTM Queue	1000000	7	210.675	2.42318	None
RTM Queue	1000000	8	198.025	1.51065	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Queue	1000000	1	374.2	0.716115	Low
RTM Queue	1000000	2	295.025	3.28556	Low
RTM Queue	1000000	3	265.725	2.18972	Low
RTM Queue	1000000	4	255.725	3.8062	Low
RTM Queue	1000000	5	247.425	3.99679	Low
RTM Queue	1000000	6	236.575	1.80455	Low
RTM Queue	1000000	7	232.55	1.56893	Low
RTM Queue	1000000	8	233.775	1.02532	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Queue	1000000	1	864.975	1	Medium
RTM Queue	1000000	2	526.225	1.09778	Medium
RTM Queue	1000000	3	396.55	4.37944	Medium
RTM Queue	1000000	4	338.875	1.40512	Medium
RTM Queue	1000000	5	366.25	9.07518	Medium
RTM Queue	1000000	6	322.25	1.45002	Medium
RTM Queue	1000000	7	289.775	2.34794	Medium
RTM Queue	1000000	8	273.2	2.68901	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Queue	1000000	1	1997.9	1.82574	High
RTM Queue	1000000	2	1099.12	1.22997	High
RTM Queue	1000000	3	780.825	1.98068	High
RTM Queue	1000000	4	623.725	1.27098	High
RTM Queue	1000000	5	750.375	20.2073	High
RTM Queue	1000000	6	648.125	6.65448	High
RTM Queue	1000000	7	566.525	1.65638	High
RTM Queue	1000000	8	515.575	2.40192	High

HTMQueue

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Queue	1000000	1	141.175	0.423659	None
TM Queue	1000000	2	260.325	0.83205	None
TM Queue	1000000	3	243.4	8.11772	None
TM Queue	1000000	4	289.425	0.83205	None
TM Queue	1000000	5	305.625	5.06116	None
TM Queue	1000000	6	322.2	2.3315	None
TM Queue	1000000	7	354.475	2.27021	None
TM Queue	1000000	8	401.675	1.54422	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Queue	1000000	1	410.075	0.27735	Low
TM Queue	1000000	2	315.9	1.19829	Low
TM Queue	1000000	3	373.25	2	Low
TM Queue	1000000	4	415	3.50823	Low
TM Queue	1000000	5	419.95	3.94838	Low
TM Queue	1000000	6	430.725	2.28148	Low
TM Queue	1000000	7	462.825	2.96561	Low
TM Queue	1000000	8	516.15	2.40725	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Queue	1000000	1	903	0	Medium
TM Queue	1000000	2	532.75	0.905822	Medium
TM Queue	1000000	3	415.3	0.9337	Medium
TM Queue	1000000	4	456.575	3.71069	Medium
TM Queue	1000000	5	541.65	2.81024	Medium
TM Queue	1000000	6	507.225	3.8796	Medium
TM Queue	1000000	7	512	5.1341	Medium
TM Queue	1000000	8	533.375	2.55704	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Queue	1000000	1	2035	0	High
TM Queue	1000000	2	1106.75	0.9337	High
TM Queue	1000000	3	792	0.960769	High
TM Queue	1000000	4	650.025	0.974022	High
TM Queue	1000000	5	813.45	31.1119	High
TM Queue	1000000	6	714.325	7.50897	High
TM Queue	1000000	7	647.2	6.40513	High
TM Queue	1000000	8	626.075	2.74095	High

STMQueue

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Queue	1000000	1	256.925	1.87425	None
STM Queue	1000000	2	357.7	5.3637	None
STM Queue	1000000	3	253.825	3.5482	None
STM Queue	1000000	4	261.925	1.67179	None
STM Queue	1000000	5	247.35	1.41421	None
STM Queue	1000000	6	245.475	1.62512	None
STM Queue	1000000	7	244.275	1.71718	None
STM Queue	1000000	8	243.425	2.28148	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Queue	1000000	1	482.05	1.88108	Low
STM Queue	1000000	2	406.675	4.79583	Low
STM Queue	1000000	3	292.725	6.4748	Low
STM Queue	1000000	4	252.9	1.85362	Low
STM Queue	1000000	5	257.65	2.24179	Low
STM Queue	1000000	6	242.725	3.52646	Low
STM Queue	1000000	7	243.075	1.70219	Low
STM Queue	1000000	8	244.425	1.1209	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Queue	1000000	1	840.55	1.67944	Medium
STM Queue	1000000	2	575.95	4.24264	Medium
STM Queue	1000000	3	406.675	1.99358	Medium
STM Queue	1000000	4	323.25	16.04	Medium
STM Queue	1000000	5	355.75	4.50641	Medium
STM Queue	1000000	6	309.2	1.1547	Medium
STM Queue	1000000	7	278.625	1.96769	Medium
STM Queue	1000000	8	251.9	4.17563	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Queue	1000000	1	1618.58	3.8796	High
STM Queue	1000000	2	976.35	4.58537	High
STM Queue	1000000	3	686.45	1.50214	High
STM Queue	1000000	4	558.4	34.2465	High
STM Queue	1000000	5	623.675	19.5363	High
STM Queue	1000000	6	549.675	3.35506	High
STM Queue	1000000	7	482.1	1.51911	High
STM Queue	1000000	8	438.5	2.86446	High

LockfreeQueue

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Lockfree Queue	1000000	1	38.05	0.226455	None
Lockfree Queue	1000000	2	159.8	7.62923	None
Lockfree Queue	1000000	3	167.425	0.862316	None
Lockfree Queue	1000000	4	162.35	0.847319	None
Lockfree Queue	1000000	5	153.75	4.44338	None
Lockfree Queue	1000000	6	174.575	6.04046	None
Lockfree Queue	1000000	7	203.625	2.46514	None
Lockfree Queue	1000000	8	232.5	11.8192	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Lockfree Queue	1000000	1	248	0	Low
Lockfree Queue	1000000	2	170.575	0.891556	Low
Lockfree Queue	1000000	3	141.675	1.09778	Low
Lockfree Queue	1000000	4	161.575	0.919866	Low
Lockfree Queue	1000000	5	165.775	2.94827	Low
Lockfree Queue	1000000	6	149.875	1.60927	Low
Lockfree Queue	1000000	7	151.05	1.30089	Low
Lockfree Queue	1000000	8	174.225	1.45884	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Lockfree Queue	1000000	1	600.275	0.531085	Medium
Lockfree Queue	1000000	2	351.175	0.423659	Medium
Lockfree Queue	1000000	3	250.475	0.697982	Medium
Lockfree Queue	1000000	4	200	0	Medium
Lockfree Queue	1000000	5	253.45	4.05728	Medium
Lockfree Queue	1000000	6	220.4	1.6641	Medium
Lockfree Queue	1000000	7	194.75	2.32048	Medium
Lockfree Queue	1000000	8	171.975	3.42315	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Lockfree Queue	1000000	1	1368	0	High
Lockfree Queue	1000000	2	749.25	0.50637	High
Lockfree Queue	1000000	3	529.375	0.620174	High
Lockfree Queue	1000000	4	422	0	High
Lockfree Queue	1000000	5	508.775	18.6142	High
Lockfree Queue	1000000	6	449.55	1.33973	High
Lockfree Queue	1000000	7	395.05	2.37508	High
Lockfree Queue	1000000	8	354.775	3.06761	High

SpinLockQueue

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Spinlock Queue	1000000	1	122.65	0.816497	None
Spinlock Queue	1000000	2	182.925	3.6899	None
Spinlock Queue	1000000	3	177.925	1.79029	None
Spinlock Queue	1000000	4	150.775	1.29099	None
Spinlock Queue	1000000	5	127.175	0.57735	None
Spinlock Queue	1000000	6	121.425	1.73205	None
Spinlock Queue	1000000	7	126.925	1.1209	None
Spinlock Queue	1000000	8	135.825	2.99144	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Spinlock Queue	1000000	1	351	0	Low
Spinlock Queue	1000000	2	300.2	1.79743	Low
Spinlock Queue	1000000	3	224.575	1.52753	Low
Spinlock Queue	1000000	4	191.625	0.83205	Low
Spinlock Queue	1000000	5	180.025	0.660225	Low
Spinlock Queue	1000000	6	159.675	0.862316	Low
Spinlock Queue	1000000	7	143.125	1.36814	Low
Spinlock Queue	1000000	8	133.275	3.83305	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Spinlock Queue	1000000	1	702.975	6.61195	Medium
Spinlock Queue	1000000	2	488.075	7.15578	Medium
Spinlock Queue	1000000	3	348.675	1.94145	Medium
Spinlock Queue	1000000	4	279.65	0.905822	Medium
Spinlock Queue	1000000	5	292.075	3.15822	Medium
Spinlock Queue	1000000	6	253	1.63299	Medium
Spinlock Queue	1000000	7	222.35	2.69853	Medium
Spinlock Queue	1000000	8	201.525	3.85307	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Spinlock Queue	1000000	1	1464	0	High
Spinlock Queue	1000000	2	881.275	1.80455	High
Spinlock Queue	1000000	3	624.2	0.877058	High
Spinlock Queue	1000000	4	496.7	0.960769	High
Spinlock Queue	1000000	5	557.6	15.525	High
Spinlock Queue	1000000	6	486.95	6.0553	High
Spinlock Queue	1000000	7	428.25	1.7097	High
Spinlock Queue	1000000	8	384.175	2.84199	High

C Histogram Measurements

HistogramSerial

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Sequential Histogram	1000000	1	26	0	None
Sequential Histogram	1000000	2	26	0	None
Sequential Histogram	1000000	3	26	0	None
Sequential Histogram	1000000	4	26	0	None
Sequential Histogram	1000000	5	26	0	None
Sequential Histogram	1000000	6	26	0	None
Sequential Histogram	1000000	7	26	0	None
Sequential Histogram	1000000	8	26	0	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Sequential Histogram	1000000	1	242	0	Low
Sequential Histogram	1000000	2	242	0	Low
Sequential Histogram	1000000	3	241.975	1	Low
Sequential Histogram	1000000	4	241.475	0.697982	Low
Sequential Histogram	1000000	5	242	0	Low
Sequential Histogram	1000000	6	242	0	Low
Sequential Histogram	1000000	7	242	0	Low
Sequential Histogram	1000000	8	242	0	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Sequential Histogram	1000000	1	599	0	Medium
Sequential Histogram	1000000	2	599	0	Medium
Sequential Histogram	1000000	3	599	0	Medium
Sequential Histogram	1000000	4	599	0	Medium
Sequential Histogram	1000000	5	599	0	Medium
Sequential Histogram	1000000	6	599	0	Medium
Sequential Histogram	1000000	7	599	0	Medium
Sequential Histogram	1000000	8	599	0	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Sequential Histogram	1000000	1	1365.78	0.891556	High
Sequential Histogram	1000000	2	1365.7	0.847319	High
Sequential Histogram	1000000	3	1365.78	0.891556	High
Sequential Histogram	1000000	4	1365.72	0.862316	High
Sequential Histogram	1000000	5	1365.58	0.767948	High
Sequential Histogram	1000000	6	1365.75	0.877058	High
Sequential Histogram	1000000	7	1365.67	0.83205	High
Sequential Histogram	1000000	8	1365.78	0.891556	High

HistogramRTM

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Histogram	1000000	1	34	0	None
RTM Histogram	1000000	2	126.675	2.79652	None
RTM Histogram	1000000	3	117.175	1.05003	None
RTM Histogram	1000000	4	149.225	0.733799	None
RTM Histogram	1000000	5	159.675	1.27098	None
RTM Histogram	1000000	6	165.175	1.94145	None
RTM Histogram	1000000	7	186.225	3.00853	None
RTM Histogram	1000000	8	192.975	3.06761	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Histogram	1000000	1	248.6	0.784465	Low
RTM Histogram	1000000	2	164.9	0.960769	Low
RTM Histogram	1000000	3	159.625	0.947331	Low
RTM Histogram	1000000	4	158.275	3.89938	Low
RTM Histogram	1000000	5	178.675	1.67179	Low
RTM Histogram	1000000	6	163.95	1.82574	Low
RTM Histogram	1000000	7	142.075	2.29269	Low
RTM Histogram	1000000	8	176.775	2.77812	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Histogram	1000000	1	606	0	Medium
RTM Histogram	1000000	2	340	0.226455	Medium
RTM Histogram	1000000	3	255.7	0.905822	Medium
RTM Histogram	1000000	4	208.925	1	Medium
RTM Histogram	1000000	5	260.125	3.8062	Medium
RTM Histogram	1000000	6	233.925	1.42325	Medium
RTM Histogram	1000000	7	210.875	3.20656	Medium
RTM Histogram	1000000	8	188.8	4.31455	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Histogram	1000000	1	1376	0	High
RTM Histogram	1000000	2	736.125	0.358057	High
RTM Histogram	1000000	3	531.575	0.767948	High
RTM Histogram	1000000	4	426.025	0.160128	High
RTM Histogram	1000000	5	509.125	22.7591	High
RTM Histogram	1000000	6	454.475	1.76141	High
RTM Histogram	1000000	7	401.575	2.40192	High
RTM Histogram	1000000	8	362.175	3.48991	High

HistogramTM

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Histogram	1000000	1	60	0	None
TM Histogram	1000000	2	128.675	0.862316	None
TM Histogram	1000000	3	116.05	1.63299	None
TM Histogram	1000000	4	126.925	1.42325	None
TM Histogram	1000000	5	155.575	1.99358	None
TM Histogram	1000000	6	178.425	3.51918	None
TM Histogram	1000000	7	205.3	2.62141	None
TM Histogram	1000000	8	240.725	1.34926	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Histogram	1000000	1	277	0	Low
TM Histogram	1000000	2	160.075	0.27735	Low
TM Histogram	1000000	3	188.775	1.76141	Low
TM Histogram	1000000	4	217.95	3.00427	Low
TM Histogram	1000000	5	246.775	3.83305	Low
TM Histogram	1000000	6	257.625	2.59684	Low
TM Histogram	1000000	7	280.125	1.25064	Low
TM Histogram	1000000	8	302.75	2.26455	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Histogram	1000000	1	634.925	0.974022	Medium
TM Histogram	1000000	2	340.325	0.57735	Medium
TM Histogram	1000000	3	250.975	1	Medium
TM Histogram	1000000	4	226.475	0.891556	Medium
TM Histogram	1000000	5	328.25	3.67249	Medium
TM Histogram	1000000	6	310.875	3.6899	Medium
TM Histogram	1000000	7	317.025	3.47519	Medium
TM Histogram	1000000	8	343.4	2.60177	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Histogram	1000000	1	1433.08	0.358057	High
TM Histogram	1000000	2	741.3	0.5547	High
TM Histogram	1000000	3	528	0	High
TM Histogram	1000000	4	422.05	0.226455	High
TM Histogram	1000000	5	541.25	11.413	High
TM Histogram	1000000	6	473.45	8.28034	High
TM Histogram	1000000	7	422.1	2.08782	High
TM Histogram	1000000	8	388.1	2.42846	High

HistogramSTM

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Histogram	1000000	1	159.775	2.54699	None
STM Histogram	1000000	2	222.25	0.50637	None
STM Histogram	1000000	3	227.375	1.79029	None
STM Histogram	1000000	4	239.325	5.3036	None
STM Histogram	1000000	5	234.6	4.64095	None
STM Histogram	1000000	6	238.425	5.97216	None
STM Histogram	1000000	7	245.375	8.05908	None
STM Histogram	1000000	8	251.6	3.78932	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Histogram	1000000	1	377.55	1.19829	Low
STM Histogram	1000000	2	246.625	1.88788	Low
STM Histogram	1000000	3	224.15	0.679366	Low
STM Histogram	1000000	4	244.875	2.47552	Low
STM Histogram	1000000	5	260.3	2.66025	Low
STM Histogram	1000000	6	265.4	1.56893	Low
STM Histogram	1000000	7	273.6	1.96116	Low
STM Histogram	1000000	8	281.9	2.2188	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Histogram	1000000	1	740.15	0.679366	Medium
STM Histogram	1000000	2	436.95	1.5359	Medium
STM Histogram	1000000	3	309.075	0.27735	Medium
STM Histogram	1000000	4	248.55	0.751068	Medium
STM Histogram	1000000	5	312.325	5.79345	Medium
STM Histogram	1000000	6	276.925	1.59326	Medium
STM Histogram	1000000	7	264.7	1.45002	Medium
STM Histogram	1000000	8	277.575	2.73158	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Histogram	1000000	1	1516.42	1.27098	High
STM Histogram	1000000	2	835.1	0.320256	High
STM Histogram	1000000	3	590.55	0.751068	High
STM Histogram	1000000	4	469.25	0.50637	High
STM Histogram	1000000	5	573.575	12.1666	High
STM Histogram	1000000	6	495.1	6.19346	High
STM Histogram	1000000	7	434.675	1.9282	High
STM Histogram	1000000	8	396.025	3.56263	High

HistogramSpinLock

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
SpinLock Histogram	1000000	1	27	0	None
SpinLock Histogram	1000000	2	74.2	4.43182	None
SpinLock Histogram	1000000	3	124.25	4.13242	None
SpinLock Histogram	1000000	4	127.9	5.45377	None
SpinLock Histogram	1000000	5	128.3	3.58057	None
SpinLock Histogram	1000000	6	134.775	5.92474	None
SpinLock Histogram	1000000	7	154.7	5.4018	None
SpinLock Histogram	1000000	8	176.825	7.08375	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
SpinLock Histogram	1000000	1	245.025	0.160128	Low
SpinLock Histogram	1000000	2	148.95	0.987096	Low
SpinLock Histogram	1000000	3	108.025	0.160128	Low
SpinLock Histogram	1000000	4	128.525	0.767948	Low
SpinLock Histogram	1000000	5	153.125	1.88788	Low
SpinLock Histogram	1000000	6	158.05	3.39683	Low
SpinLock Histogram	1000000	7	165.625	3.30113	Low
SpinLock Histogram	1000000	8	156.475	2.29269	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
SpinLock Histogram	1000000	1	603.875	0.947331	Medium
SpinLock Histogram	1000000	2	328	0	Medium
SpinLock Histogram	1000000	3	233	0	Medium
SpinLock Histogram	1000000	4	186.9	0.960769	Medium
SpinLock Histogram	1000000	5	228.875	6.61971	Medium
SpinLock Histogram	1000000	6	201.375	1.38675	Medium
SpinLock Histogram	1000000	7	179.45	2.53185	Medium
SpinLock Histogram	1000000	8	162.05	1.45002	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
SpinLock Histogram	1000000	1	1368	0	High
SpinLock Histogram	1000000	2	725	0	High
SpinLock Histogram	1000000	3	515	0	High
SpinLock Histogram	1000000	4	410	0	High
SpinLock Histogram	1000000	5	503.6	15.2904	High
SpinLock Histogram	1000000	6	437	2.92645	High
SpinLock Histogram	1000000	7	386.4	3.34357	High
SpinLock Histogram	1000000	8	349.85	3.35888	High

D Refactored Histogram Measurements

HistogramSerial

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Sequential Histogram	1000000	1	26	0	None
Sequential Histogram	1000000	2	26	0	None
Sequential Histogram	1000000	3	26.025	0.160128	None
Sequential Histogram	1000000	4	26	0	None
Sequential Histogram	1000000	5	26	0	None
Sequential Histogram	1000000	6	26	0	None
Sequential Histogram	1000000	7	26	0	None
Sequential Histogram	1000000	8	26	0	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Sequential Histogram	1000000	1	242	0	Low
Sequential Histogram	1000000	2	242	0	Low
Sequential Histogram	1000000	3	242	0	Low
Sequential Histogram	1000000	4	242	0	Low
Sequential Histogram	1000000	5	242.025	0.160128	Low
Sequential Histogram	1000000	6	242	0	Low
Sequential Histogram	1000000	7	242	0	Low
Sequential Histogram	1000000	8	242	0	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Sequential Histogram	1000000	1	599	0	Medium
Sequential Histogram	1000000	2	599	0	Medium
Sequential Histogram	1000000	3	599	0	Medium
Sequential Histogram	1000000	4	599	0	Medium
Sequential Histogram	1000000	5	599	0	Medium
Sequential Histogram	1000000	6	599	0	Medium
Sequential Histogram	1000000	7	599	0	Medium
Sequential Histogram	1000000	8	599	0	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
Sequential Histogram	1000000	1	1367.65	0.816497	High
Sequential Histogram	1000000	2	1367.47	0.697982	High
Sequential Histogram	1000000	3	1367.6	0.784465	High
Sequential Histogram	1000000	4	1367.72	0.862316	High
Sequential Histogram	1000000	5	1367.55	0.751068	High
Sequential Histogram	1000000	6	1367.58	0.767948	High
Sequential Histogram	1000000	7	1367.62	0.800641	High
Sequential Histogram	1000000	8	1367.53	0.733799	High

HistogramRTM

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Histogram	1000000	1	34	0	None
RTM Histogram	1000000	2	66.575	1.52753	None
RTM Histogram	1000000	3	79.275	5.07129	None
RTM Histogram	1000000	4	128.1	6.05953	None
RTM Histogram	1000000	5	129.025	9.94472	None
RTM Histogram	1000000	6	145.925	3.70377	None
RTM Histogram	1000000	7	153.575	2.28148	None
RTM Histogram	1000000	8	165.2	4.10128	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Histogram	1000000	1	249	0	Low
RTM Histogram	1000000	2	152.275	0.531085	Low
RTM Histogram	1000000	3	118.2	0.50637	Low
RTM Histogram	1000000	4	105.775	0.947331	Low
RTM Histogram	1000000	5	116.2	2.28709	Low
RTM Histogram	1000000	6	101.5	1.50214	Low
RTM Histogram	1000000	7	97.275	1.47631	Low
RTM Histogram	1000000	8	119.95	3.05505	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Histogram	1000000	1	609	0	Medium
RTM Histogram	1000000	2	335.725	0.862316	Medium
RTM Histogram	1000000	3	239.55	0.847319	Medium
RTM Histogram	1000000	4	193.95	1.1547	Medium
RTM Histogram	1000000	5	232.65	3.6162	Medium
RTM Histogram	1000000	6	202	1.13228	Medium
RTM Histogram	1000000	7	180.9	2.73627	Medium
RTM Histogram	1000000	8	162.575	2.0318	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
RTM Histogram	1000000	1	1377.15	0.392232	High
RTM Histogram	1000000	2	733.95	0.987096	High
RTM Histogram	1000000	3	523	0	High
RTM Histogram	1000000	4	418.025	0.160128	High
RTM Histogram	1000000	5	507.825	20.145	High
RTM Histogram	1000000	6	443.55	7.60904	High
RTM Histogram	1000000	7	392.575	2.22457	High
RTM Histogram	1000000	8	357.85	3.17845	High

HistogramTM

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Histogram	1000000	1	60	0	None
TM Histogram	1000000	2	86.2	0.452911	None
TM Histogram	1000000	3	90.625	1.96769	None
TM Histogram	1000000	4	106.95	1.82574	None
TM Histogram	1000000	5	115.325	1.18754	None
TM Histogram	1000000	6	126.825	1.36814	None
TM Histogram	1000000	7	142.55	1.2195	None
TM Histogram	1000000	8	163.825	1.07417	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Histogram	1000000	1	277	0	Low
TM Histogram	1000000	2	157	0	Low
TM Histogram	1000000	3	137.675	1.51065	Low
TM Histogram	1000000	4	159.4	4.0762	Low
TM Histogram	1000000	5	188.025	2.10616	Low
TM Histogram	1000000	6	186	0.987096	Low
TM Histogram	1000000	7	197.35	0.9337	Low
TM Histogram	1000000	8	211.4	1.08604	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Histogram	1000000	1	643.175	0.423659	Medium
TM Histogram	1000000	2	336.05	0.226455	Medium
TM Histogram	1000000	3	246.625	0.800641	Medium
TM Histogram	1000000	4	218.175	0.697982	Medium
TM Histogram	1000000	5	259.95	5.27208	Medium
TM Histogram	1000000	6	230.75	1.78311	Medium
TM Histogram	1000000	7	220.1	1.37747	Medium
TM Histogram	1000000	8	231.65	2.47034	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
TM Histogram	1000000	1	1416	0	High
TM Histogram	1000000	2	737.4	0.640513	High
TM Histogram	1000000	3	525	0	High
TM Histogram	1000000	4	421	0	High
TM Histogram	1000000	5	509.825	23.6984	High
TM Histogram	1000000	6	448.975	5.06116	High
TM Histogram	1000000	7	396.4	0.877058	High
TM Histogram	1000000	8	367.275	1.18754	High

HistogramSTM

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Histogram	1000000	1	172.475	1.18754	None
STM Histogram	1000000	2	141.45	0.716115	None
STM Histogram	1000000	3	117.775	1	None
STM Histogram	1000000	4	102.85	1.03775	None
STM Histogram	1000000	5	98.8	1.13228	None
STM Histogram	1000000	6	89.575	0.83205	None
STM Histogram	1000000	7	82.425	1.44115	None
STM Histogram	1000000	8	77.55	1.83973	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Histogram	1000000	1	389.8	3.50092	Low
STM Histogram	1000000	2	236.95	0.987096	Low
STM Histogram	1000000	3	176.225	0.733799	Low
STM Histogram	1000000	4	145.5	0.716115	Low
STM Histogram	1000000	5	168.45	4.23054	Low
STM Histogram	1000000	6	147.75	0.9337	Low
STM Histogram	1000000	7	131.75	1.28103	Low
STM Histogram	1000000	8	121.725	2.71274	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Histogram	1000000	1	751.375	0.974022	Medium
STM Histogram	1000000	2	429.675	0.83205	Medium
STM Histogram	1000000	3	308.025	0.160128	Medium
STM Histogram	1000000	4	249.775	16.9274	Medium
STM Histogram	1000000	5	291.625	5.57697	Medium
STM Histogram	1000000	6	253.1	1.41421	Medium
STM Histogram	1000000	7	223.25	1.32045	Medium
STM Histogram	1000000	8	205.125	3.22252	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
STM Histogram	1000000	1	1585.17	1.33012	High
STM Histogram	1000000	2	878.525	98.749	High
STM Histogram	1000000	3	637.1	87.8232	High
STM Histogram	1000000	4	489.125	33.7156	High
STM Histogram	1000000	5	570.275	12.2213	High
STM Histogram	1000000	6	492.7	1.61722	High
STM Histogram	1000000	7	433.225	1.98068	High
STM Histogram	1000000	8	396.425	2.80567	High

HistogramSpinLock

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
SpinLock Histogram	1000000	1	29	0	None
SpinLock Histogram	1000000	2	51.025	0.27735	None
SpinLock Histogram	1000000	3	48.75	0.877058	None
SpinLock Histogram	1000000	4	45	0	None
SpinLock Histogram	1000000	5	39.4	0.640513	None
SpinLock Histogram	1000000	6	36.65	1.1547	None
SpinLock Histogram	1000000	7	35.425	2.70327	None
SpinLock Histogram	1000000	8	35.425	3.08429	None

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
SpinLock Histogram	1000000	1	247	0	Low
SpinLock Histogram	1000000	2	144.575	0.767948	Low
SpinLock Histogram	1000000	3	104.025	0.160128	Low
SpinLock Histogram	1000000	4	85.025	0.160128	Low
SpinLock Histogram	1000000	5	102.3	3.00427	Low
SpinLock Histogram	1000000	6	89.75	1.79743	Low
SpinLock Histogram	1000000	7	81.975	2.1543	Low
SpinLock Histogram	1000000	8	74.525	2.61652	Low

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
SpinLock Histogram	1000000	1	604	0	Medium
SpinLock Histogram	1000000	2	329	0	Medium
SpinLock Histogram	1000000	3	233	0.226455	Medium
SpinLock Histogram	1000000	4	185.975	1	Medium
SpinLock Histogram	1000000	5	227.925	3.30113	Medium
SpinLock Histogram	1000000	6	198.85	1.26085	Medium
SpinLock Histogram	1000000	7	178.425	2.71274	Medium
SpinLock Histogram	1000000	8	162.125	3.04243	Medium

Datastructure	Elements	Threads	Mean	StandardDeviation	Workload
SpinLock Histogram	1000000	1	1370.47	0.697982	High
SpinLock Histogram	1000000	2	725.025	0.160128	High
SpinLock Histogram	1000000	3	514	0	High
SpinLock Histogram	1000000	4	410	0	High
SpinLock Histogram	1000000	5	504.9	9.42174	High
SpinLock Histogram	1000000	6	435.925	2.10616	High
SpinLock Histogram	1000000	7	384.825	2.40192	High
SpinLock Histogram	1000000	8	350.35	3.28165	High

Eidesstattliche Erklärung

Ich, Philipp Schoppe, Matrikel-Nr. 630508, versichere hiermit an Eides statt durch meine eigene Unterschrift, dass ich die vorstehende Arbeit mit dem Thema

> *Evaluation of Transactional Memory and Other Techniques to Improve the Performance of Algorithms in High Energy Physics for New Processor Architectures*

selbstständig und ohne unzulässige fremde Hilfe angefertigt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Erklärung bezieht sich auch auf in der Arbeit gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen. Diese Arbeit wurde bisher in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Steinfurt, den 17.11.2014

Philipp Schoppe