Practical Solutions for Multicore Programming
From Crafting a High-Performance Ready-to-Go STM
to non-Linearizable Data Structures

Thesis submitted for the degree of Doctor of Philosophy
by
Guy Korland

This work was carried out under the supervision of
Professor Nir Shavit

Submitted to the Senate of Tel Aviv University
July 2011
To my dear wife.
Acknowledgements

First and foremost, I would like to thank my advisor, Professor Nir Shavit. Nir gave me a blank check to run with every initiative I thought of, while providing me every intellectual support needed and an amazing pool of resources. I wish to thank Professor Yehuda Afek for the opportunity to work with him and his amazing group. It is also an opportunity and a pleasure to thank my other co-authors and collaborators: Maria Natanzon, Eitan Yanovsky and Arie Zilberstein; this amazing group of students helped me carry my ideas from fantasy into reality. I also would like to thank Professor Pascal Felber for helping me in developing Deuce and pushing it to become a known research tool.

I appreciate the fruitful discussions with my lab members and I thank them for the time we spent together. I would especially like to mention Moran Tzafrir, of blessed memory. It was a great pleasure to exchange ideas for hours with Moran; her presence for sure will be missed.

I wish to thank my dear parents for their love, encouragement and on-going support over the years. Without them pushing me from a young age, even though I did not want to be pushed, I would never have gotten to where I am. Last but not least, I wish to thank my dearest wife Hila, for her love and friendship, and for keeping up with me while I chase my dreams. This thesis is dedicated to her.
Abstract

This dissertation presents several novel techniques for practical multicore programming in a shared memory multicore machine.

The basis of the research initiative, which is covered in the first part of this dissertation, is DEUCE [50], a novel open-source Java\textsuperscript{TM} framework for Software Transactional Memory (STM). DEUCE has several desired features not found in earlier Java\textsuperscript{TM} STM frameworks. When this research was started there did not exist an efficient full-featured Java\textsuperscript{TM} STM framework that could be added to an existing application without changes to its compiler or libraries. It seemed that any efficient Java\textsuperscript{TM} STM would require compiler support.

DEUCE is intended to fill this gap. It is non-intrusive in the sense that no modifications to the Java\textsuperscript{TM} virtual machine (JVM) or extensions to the language are necessary. It uses, by default, an original locking design that detects conflicts at the level of individual fields without a significant increase in the memory footprint or Garbage Collection (GC) overhead. This locking scheme provides finer granularity and better parallelism than former object-based lock designs. It also supports a pluggable STM back end and an open easily extendable architecture, allowing researchers to integrate and test their own STM algorithms. Finally, DEUCE supports novel compiler optimization techniques [4] in an attempt to drive the STM overhead lower.

The second part of this dissertation deals with the design of concurrent data structures by weakening the consistency assumptions. First, we present the ED-Tree [2], a pool structure based on a combination of the elimination-tree and diffracting-tree paradigms, allowing high degrees of parallelism with reduced contention.

Then, we deal with Linearizability, the main correctness condition that most concurrent object implementations comply with, imposes tight synchronization between the object’s concurrent operations. This tight synchronization usually comes with a performance and scalability price. Yet, these implementations are often employed in an environment where a more relaxed consistency model suffices, where strict linearizability is not a must. As an alternative we provide a quantitative definition of limited
non-determinism, a notion we call \textit{Quasi Linearizability} \cite{3}. Roughly speaking, an implementation of an object is quasi linearizable if each run of the implementation is at a bounded “distance” from some linear run of the object.
# Contents

## 1 Introduction

1.1 Overview ........................................ 2
   1.1.1 Programming Methods .......................... 3

1.2 Transactional Memory ............................... 4
   1.2.1 Hardware Transactional Memory ................. 4
   1.2.2 Software Transactional Memory .................. 4
   1.2.3 STM Compiler Optimizations .................... 8
   1.2.4 Contribution .................................. 10

1.3 Non-Linearizable Objects .......................... 11
   1.3.1 Producer-Consumer Pools ....................... 12

1.4 Outline ......................................... 18

## 2 Noninvasive Concurrency with a Java$^\text{TM}$ STM

2.1 Overview ....................................... 20
   2.1.1 Concurrent Programming with DEUCE ............ 21
   2.1.2 The DEUCE Runtime ............................ 23

2.2 The DEUCE Implementation .......................... 23
   2.2.1 Overview ..................................... 23
   2.2.2 Instrumentation Framework ...................... 24
   2.2.3 Summary ...................................... 28
   2.2.4 Optimizations ................................ 28

2.3 Customizing Concurrency Control .................... 29
   2.3.1 TL2 Context ................................ 30
   2.3.2 LSA Context ................................ 31
4.3.1 The tree nodes ........................................ 77
4.3.2 Adaptivity ............................................. 79
4.3.3 An Unfair Synchronous Pool [47] .................. 80
4.3.4 An Object Pool ....................................... 80
4.3.5 Starvation Avoidance ................................. 81
4.4 Performance Evaluation ................................. 81
4.5 Conclusions ............................................. 85

5 Quasi-Linearizability ..................................... 87
  5.1 Overview ............................................... 88
    5.1.1 Consistency Models ................................. 90
  5.2 The Definition of Quasi Linearizability .......... 92
    5.2.1 Lineazability Review .............................. 92
    5.2.2 Quasi-Linearizability ............................ 94
    5.2.3 Definition details ................................. 97
  5.3 Implementations ...................................... 102
    5.3.1 Random Dequeued Queue ......................... 102
  5.4 Segmented Queue .................................... 108
    5.4.1 Implementation ................................... 108
    5.4.2 Proof of Quasi-Linearizability ................. 112
    5.4.3 Segmented Stack .................................. 115
  5.5 Performance Evaluation ............................. 115
  5.6 Bitonic[W] Counting Network ...................... 116
  5.7 Conclusions ......................................... 118

Bibliography .............................................. 119
List of Tables

2.1 Memory footprint and processing overhead ........................................ 33
2.2 Comparing AtomJava, DSTM2, and Deuce ..................................... 39

3.1 % of reads of immutable memory. .................................................. 45
3.2 % of synchronization operations removed. ....................................... 47
3.3 % of transaction-local accesses detected. ....................................... 50
3.4 % of reads from memory not changed within the transaction. .......... 52
3.5 % of reads eliminated after applying PRE. ..................................... 54
3.6 % of reads from locations not written to before. ............................. 58
3.7 % of writes to locations which will not be subsequently read from. ... 59
3.8 % of reads hoisted outside of transactions. .................................... 62
List of Figures

2.1 @atomic method example. ........................................... 20
2.2 synchronized method example. ................................. 21
2.3 Atomic block example. ............................................. 21
2.4 @Atomic method example. ........................................ 22
2.5 Main components of the DEUCE architecture. .............. 24
2.6 Fields address. ..................................................... 25
2.7 Fields accessors. ................................................... 25
2.8 Duplicate method. ................................................... 26
2.9 Atomic method. ..................................................... 27
2.10 Context interface. ................................................. 30
2.11 The red/black tree benchmark (Sun). ....................... 34
2.12 The linked list benchmark (Sun). ............................. 35
2.13 The linked list benchmark (Azul). ............................ 35
2.14 The skiplist benchmark (Sun). ................................. 35
2.15 The skiplist benchmark (Azul). ............................... 36
2.16 The Lee Routing benchmark (Sun). ........................... 37
2.17 The Cache4J benchmark (Sun). ............................... 37
2.18 The Vacation benchmark from the STAMP suite (Sun). ... 38

3.1 Immutable field access. ............................................. 44
3.2 Redundant load. .................................................... 52
3.3 Eliminate redundant load. ....................................... 52
3.4 Redundant writes. .................................................. 54
3.5 Eliminate redundant writes. .................................... 55
3.6 Transfer function $\tau$. ......................................... 57
3.7 Tightening transaction scope. .................................. 60
3.8 Microbenchmarks comparative results. (higher is better) ............ 64
3.9 STAMP Benchmarks comparative results. (lower is better) .......... 66
4.1 A Balancer. ......................................................... 70
4.2 A diffracting tree obeys the step property. .......................... 71
4.3 A Tree[4] leading to 4 lock-free queues. ............................ 74
4.4 Full ED-Tree. ......................................................... 76
4.5 Balancer structure. ................................................. 77
4.6 Toggle bit. .............................................................. 77
4.7 Exchanger structure. ............................................... 78
4.8 Unfair synchronous queue benchmark. ............................... 82
4.9 Throughput of blockingQueue object pool implementations. .... 82
4.10 Throughput of a synchronous queue as the work load changes. .. 83
4.11 Performance of Linked-Blocking queue and Synchronous queue. .. 84
4.12 Performance changes of Resource pool and Unfair SynchronousQueue. 84
4.13 Elimination rate by levels as concurrency increases. ............... 85
4.14 Elimination range as the work load changes for 64 threads. ....... 85
5.1 Random Dequeue Queue ............................................. 102
5.2 Segmented Queue ................................................... 109
5.3 Segmented queue definitions ....................................... 109
5.4 Concurrent queue benchmark results ................................ 116
5.5 Bitonic[4] Counting Network ....................................... 117
Chapter 1

Introduction
1.1 Overview

Moore’s law [62] has retained its relevance for the last-half century. It defines a rule that predicts the number of transistors on an integrated circuit. According to the law, the number of transistors on an integrated circuit will be doubled every two years and this trend is expected to hold true until the year 2020.

Nevertheless, despite the fact that Moore’s law is strictly adhered to by CPU manufacturers, well-established benchmarks have not shown any major performance improvement in the last five years. These puzzling results point to a major shift in CPU design, from high frequency single core CPUs to low frequency multicore CPUs.

Multicore CPUs have become commonplace, with dual-cores powering almost any modern portable or desktop computer, and quad-cores being the norm for new servers. While multicore hardware has been advancing at an accelerated pace, software for multicore platforms seems to be at a crossroads. Utilizing multicore CPUs to achieve high throughput is a non-trivial task. One must overcome several sources of bottlenecks in order to provide a scalable solution: the machine hardware resources, such as memory and computational units; communication resources such as bus traffic; and application bottlenecks.

While the effects of hardware-related bottlenecks can generally be reduced by physically enhancing the hardware, communication- and application-related bottlenecks are inherent. The large number of processors imposes higher memory access latency, and sequentially executing programs (or parts of them) do not utilize the available computational resources. A dominating trend in research of high throughput programming on shared memory multiprocessors is the design of specialized algorithms for concurrent access to shared data. These algorithms are usually structured to allow higher parallelism while keeping communication resources usage as low as possible.

When programming in a concurrent environment, one must be aware of the constantly changing state of the shared data. Converting a sequential algorithm to a concurrent one can be an extremely difficult task, but can be crucial to the program’s correctness and efficiency. There is often a tradeoff between the complexity of a program and the level of parallelism it provides. Proving concurrent algorithms correct can be tedious and time consuming due to the non-deterministic nature of parallel execution.
1.1. OVERVIEW

1.1.1 Programming Methods

Currently, two diverging programming methods are commonly used. The first exploits concurrency by synchronizing multiple threads based on locks. This approach is known to be a double-edged sword: on the one hand, locks give the programmer a fine-grained way to control the application’s critical sections, allowing an expert to provide great scalability; on the other hand, because of the risk of deadlocks, starvation, priority inversions, etc., they impose a great burden on non-expert programmers, often leading them to prefer the safer but non-scalable alternative of coarse-grained locking. Coarse-grained locking is in many cases a bad choice for the synchronization building-block of concurrent applications. The reason is that locks provide separation in time, but not in space. Using a small set of locks for large and complex data structures limits the availability of large portions of the shared data. Threads that operate on different parts of the data structure should not block each other from making progress merely because they are performing a similar operation at the same time. Coarse-grained locks rarely consider the locality of operations, only their timing. By decreasing the granularity of locks (and increasing their number), for some data structures it is possible to reduce the amount of contention. However, fine-grained locking algorithms can be quite complex.

The second method is a share-nothing model common in Web based architectures. The applications usually contain only the business logic, deployed in a container, while the state is saved in an external multi-versioned control system, such as database, message queue, or distributed cache. While this method removes the burden of handling concurrency in the application, it imposes a huge performance impact on data accesses and, for many types of applications, is difficult to apply.

Another variant of the share-nothing model is the Actor model \[44\]. Here, computational units are just endpoints exchanging immutable messages with one another, and reacting to received messages by executing a given computation. In such a concurrency model, there is no shared resource, nor there is a need for resource synchronization: messages are immutable, and each computational unit is only able to change its own state in response to a received message. Unlike the pure share-nothing model, in the Actor model the implementation usually imposes minimal impact on performance. However, it is fairly obvious that in many-core machines (thousands of cores) scalability will be impacted. Last, the Actor model major drawback and its barrier to becoming a common model is the fact that it imposes a major mind shift in software design from
CHAPTER 1. INTRODUCTION

Shared-State concurrency to a Message-Passing concurrency paradigm.

1.2 Transactional Memory

The hope is that transactional memory (TM) \cite{herlihy1993transactional} will simplify concurrent programming by combining the desirable features of both methods, providing state-aware shared memory with simple concurrency control.

1.2.1 Hardware Transactional Memory

In 1993, Herlihy and Moss \cite{herlihy1993transactional} proposed a hardware transactional memory (HTM) system based on the cache coherency infrastructure in bus based shared memory multiprocessors. The proposed HTM architecture allowed optimistic atomic execution of bounded size transactions. However, it was never implemented, mainly due to the transaction size limitation. Since then, several hardware-based solutions \cite{herlihy2002transactional,herlihy2003transactional,herlihy2004transactional} have been suggested to cope with the transaction size limits and other drawbacks such as support for nesting and I/O operations within transactions. Additionally, two approaches for hybrid software/hardware TM have become public \cite{herlihy2005transactional,herlihy2006transactional}.

All hardware-implemented transactional memory schemes involve expanding the machine instruction set with new instructions for transaction manipulation. These instructions are either special loads and stores used within transactions or mode changing instructions, from non-transactional to transactional mode and vice versa. The hardware constructs for supporting transactions are usually additional per-processor associative memory units used for buffering the transaction’s temporary data. The transaction size limit in HTMs originates from the bounded size of that additional cache memory, while the limit on transaction duration is in many cases a result of the weak persistency of the buffered data over context switches and I/O operations.

1.2.2 Software Transactional Memory

Software Transactional Memory (STM), originally proposed in 1995 by Shavit and Touitou \cite{shavit1995software}, is an emerging approach that provides developers of concurrent software with a powerful tool: the atomic block, which aims to ease multi-threaded programming and enable more parallelism. Conceptually, statements contained in an atomic block appear to execute as a single atomic unit: either all of them take effect together, or
none of them take effect at all. In this model, the burden of carefully synchronizing concurrent access to shared memory, traditionally done using locks, semaphores, and monitors, is relieved. Instead, the developer needs only to enclose statements that access shared memory by an atomic block, and the STM implementation guarantees the atomicity of each block.

Implementing transactional memory in software has two major advantages: one is the availability of software libraries, in contrast to architectural modifications that are also extremely expensive to manufacture even as prototypes, and the other is the virtually unlimited resources available, namely memory. However, in all STMs, there is a price in latency of load and store instructions. While memory access instructions used in HTMs are in most cases identical in latency to classic loads and stores, when emulated in software, transactional reads and writes involve additional checks and address calculations for reaching the actual data. When reading from a memory location, it is essential to identify whether that location is currently participating in an active transaction, and in some cases, follow pointers to reach the actual value. An additional source of overhead is the need to traverse the read-set and/or write-set of a transaction on commit time, a task performed in parallel and as a part of the cache protocol in some hardware alternatives.

In the past several years there has been a flurry of design and implementation work of STMs. Shavit and Touitou’s proposed STM implementation as presented in [74], supported static transactions. In other words, ones where the addresses accessed by a transaction are known in advance. Their implementation is lock-free, as aborting threads (non-recursively) help the transaction that caused them to abort.

In [9], Anderson and Moir implemented lock-free multi-word CAS (MWCAS), a variation of which Moir used in [60] to build a lock-free dynamic STM. In this construction, the memory is divided into $B$ blocks whose threads lock during the execution of transactions. However, because each transaction involves applying a $B$-word MWCAS, there is an inconvenient tradeoff between efficiency and parallelism. Moir also describes a wait-free variation by applying a different helping scheme.

Herlihy et al. [39] introduced a dynamic obstruction-free STM. Its main principle is transactional objects pointing to transactional records, which, in turn, point to the object’s versions before and after the transaction. Transactions are aborted and committed by setting a status word pointed-to from each transactional record. These levels
of indirection enabled an elegant scheme that supports parallelism of transactions that do not overlap in memory access. However, the overhead of accessing memory and validating and committing transactions is relatively high. Concurrently, Harris and Fraser [32] developed an STM with a different approach, WSTM, in which memory words are associated with transactions using a map from heap addresses to transactional descriptors. The set of addresses participating in a transaction is acquired only at commit-time, minimizing the chances for collisions with other transactions. This approach requires reading-only transactions to be visible to possibly colliding modifying transactions by writing to shared memory even when the actual data is unchanged. In his Ph.D. dissertation [28] Fraser described OSTM, an STM implementation similar to WSTM, where transactions handle objects rather than individual memory words.

Marathe et al. [55] proposed Adaptive STM (ASTM), a combination of DSTM and OSTM, where objects’ association with transactions alternates from DSTM-like to OSTM-like, depending on the application. Marathe et al. [56] introduced the Rochester STM (RSTM), an improved STM implementation in which the data is co-located with transactional meta-data, reducing the overhead cost of transactional operations.

In [25], Ennals claimed that obstruction-freedom is not an essential property of STMs, and by dropping this requirement, STMs can be made to execute significantly faster. He proposed a lock-based STM similar to the obstruction-free DSTM and demonstrated performance improvements. Both Ennals’ implementation and the one Saha et al. [72] developed for their Multi-Core RunTime (McRT) use versioned locks for transactional objects, committing and validating transactions by comparing the versions of read-set addresses while write-set ones are locked.

A devious issue with STM is that unless special care is taken, only committed transactions are guaranteed to be consistent. Speculative transactions might observe an inconsistent state and only subsequently detect that they should roll back. These “zombies” can produce surprising behavior by taking impossible branches or performing transactional accesses to the wrong object. The read of a single impossible value can produce an infinite loop, so a transparent STM must either prevent inconsistent reads or instrument back edges to periodically revalidate the transaction. This correctness property was formalized by Guerraoui and Kapalka [30] as opacity. The Transactional Locking II (TL2) [22] and Lazy Snapshot Algorithm (LSA) [69] algorithms, which are described at length in subsections 2.3.1 and 2.3.2 respectively, use a global version-clock
validation to efficiently validate a transaction after each read, guaranteeing consistency for all intermediate states.

The TL2 algorithm and the LSA algorithm, proposed independently by Dice et al. and Riegel et al. respectively, show overall performance comparable to (and in many cases better than) that of all former STM algorithms, both lock-based and non-blocking. Perhaps more importantly, on various benchmarks, TL2 delivers performance that is competitive with the best hand-crafted fine-grained concurrent structures. Specifically, it is ten times faster than a single lock structure implementations.

However, the state-of-the-art STM implementation today is less than appealing. With the notable exception of transactional C/C++ compilers offered by Intel [64] and the upcoming GCC [65] 4.7 (based on TinySTM [26] which is in turn based on TL2 2.3.1), most STM initiatives have remained academic experiments, applicable only to “toy applications”. Though we have learned much from the process of developing them, they have not reached a state that enables them to be seriously field tested. There are several reasons for this. Among them are the problematic handling of many features of the target language, the large performance overheads [17], even on low load benchmarks, and the lack of support for legacy libraries. Moreover, many of the published results have been based on prototypes whose source code is unavailable or poorly maintained, making a reliable comparison between the various algorithms and designs very difficult.

Therefore, a need for an open source ground-base framework that can be used by researchers seems like a necessity. The idea of creating an open source environment that can be used for testing and comparing different algorithm has been successfully implemented several times. One example is the Jikes RVM[7]. Jikes has been used in many works to test and compare new Garbage Collection algorithms and other JVM optimizations. Another example is ImageJ [80], Java™ image processing program. ImageJ was designed with an open architecture that provides extensibility via Java™ plugins. Many custom acquisition, analysis and processing plugins were developed using ImageJs built in editor and Java compiler.

Lastly, in order to make a piece of code transactional, it usually undergoes an instrumentation process that replaces memory accesses with calls to STM library functions. These functions handle the bookkeeping of logging, committing, and rolling-back of values according to the STM protocol. Naïve instrumentation introduces redundant
function calls, for example, for values that are provably transaction-local. In addition, an STM with homogeneous implementation of its functions, while general and correct, will necessarily be less efficient than a highly-heterogeneous implementation: the latter can offer specialized functions that handle some specific cases more efficiently. For example, a homogeneous STM might offer a general \texttt{STMRead()} method, while a heterogeneous STM might offer also a specialized \texttt{STMReadThreadLocal()} method that assumes that the value read is thread-local, and as a consequence, can optimize away validations of that value.

We are aiming to answer all the requirements above. First, to deliver a full STM that can be easily (without any code change) planted in an existing application, second, to provide a framework for testing new STM algorithms, and last, to keep it efficient by minimizing its overhead.

1.2.3 STM Compiler Optimizations

In order to minimize the STM overhead, STM-specific compiler optimizations should be applied. The literature on compiler optimizations that are specific to transactional memory implementations revolves mostly around in-place-update STMs [79]. Harris et al. [34] present the baseline STM optimizations that appear in many subsequent works. Among them are: Decomposition of STM library functions to heterogeneous parts; code motion optimizations that make use of this decomposition to reduce the number of “open-for-read” operations; early upgrade of “open-for-read” into “open-for-write” operations if a write-after-read will occur; suppression of instrumentation for transaction-local objects and more. In [1], immutable objects are also exempt from instrumentation. In addition, standard compiler optimization techniques (see [48] for a full treatment), such as loop peeling, method inlining, and redundancy elimination algorithms, are applied to atomic blocks.

Eddon and Herlihy [24] apply fully interprocedural analyses to discover thread-locality and subsequent accesses to the same objects. Such discoveries are exploited for optimized “fast path” handling of the cases. Similar optimizations also appear in [23, 86]. We note that the above papers optimize for in-place-update STMs. In such an STM protocol, once a memory location is “open-for-write”, memory accesses to it are nearly transparent (free), because the memory location is exclusively owned by the transaction. Our work is different because it targets lazy-update STMs, where
subsequent instrumented accesses to memory cannot be made much cheaper than initial accesses; e.g., the write-set must still be searched on every read. We solve this problem by transforming instrumented reads and writes, which access shared memory, into uninstrumented reads and writes that access local variables.

Michael et al. [79] propose several optimizations for a TL2-like STM:

1. When multiple memory locations are read in succession, each read is instrumented such that the location is pre-validated, read, and then post-validated. By reordering the instructions so that all the pre-validations are grouped together, followed by the reads, and concluded by the post-validations, they increase the time window between memory fences, enabling the CPU to parallelize the memory reads.

2. Post-validation can sometimes be postponed as long as working with “unsafe” values can be tolerated. This eliminates or groups together expensive memory barrier operations. Shpeisman et al. [77] present barrier aggregation, a similar but simpler optimization that reuses barriers inside a basic block if they guard the same object.

Beckman et al.’s [13] work provides optimizations for thread-local, transaction-local, and immutable objects that are guided by access permissions. These are Java attributes that the programmer must use to annotate program references. For example, the @Imm attribute denotes that the associated reference variable is immutable. Access permissions are verified statically by the compiler, and then used to optimize the STM instrumentation for the affected variables.

Partial redundancy elimination (PRE) [49, 63] techniques are widely used in the field of compiler optimizations; however, most of the focus was on removing redundancies of arithmetic expressions. [27, 46] was the first work to apply PRE to Java™ access path expressions, for example, expressions like a.b[i].c. This variant of PRE is also called load elimination. As a general compiler optimization, this optimization might be unsound because it might miss concurrent updates by a different thread that changes the loaded value. Therefore, some papers [12, 85] propose analyses to detect when load elimination is valid. Scalar promotion, which eliminates redundant memory writes, was introduced by Lu and Cooper [53], and improved by later works (for example [78]).


1.2.4 Contribution

In this thesis, we introduce DEUCE, a novel open-source Java framework for transactional memory. DEUCE has several desired features not found in earlier Java™ STM frameworks. As we discuss in Section 2.1, before DEUCE there did not exist an efficient Java™ STM framework that delivered a full set of features and can be added to an existing application without changes to its compiler or libraries. It was not clear if one could build such an efficient “compiler independent” Java™ STM.

DEUCE is intended to fill this gap. It is non-intrusive in the sense that no modifications to the Java™ virtual machine (JVM) or extensions to the language are necessary. It uses, by default, an original locking design that detects conflicts at the level of individual fields without a significant increase in the memory footprint (no extra data is added to any of the classes) and therefore there is no GC overhead. This locking scheme provides finer granularity and better parallelism than former object-based lock designs.

DEUCE provides weak atomicity; in other words, it does not guarantee that concurrent accesses to a shared memory location from both inside and outside a transaction are consistent. This is in order to avoid a performance penalty when accessing a memory location outside a transaction. Supporting strong atomicity would have require instrumenting not only the code paths which are part of a transaction but also every memory access across the code.

Finally, DEUCE supports a pluggable STM back-end and an open and easily extensible architecture, allowing researchers to integrate and test their own STM algorithms within the DEUCE framework.

DEUCE has been heavily optimized for efficiency. While there is still room for improvement, our performance evaluations on several high-end machines (up to 128 hardware threads on a 16-core Sun Niagara-based machine and a 96-core Azul machine), samples of which we include in the dissertation, demonstrate that it scales well. Our benchmarks show that it outperforms the main competing JVM-independent Java™ STM, the DSTM2 framework [37], in many cases by two orders of magnitude, and in general scales well on many workloads. This thesis shows for the first time that one can actually design a Java™ STM with reasonable performance without compiler support.

The DEUCE framework has been in development for more than three years, and we believe it has reached a level of maturity sufficient to allow it to be used by developers.
with no prior expertise in transactional memory. It is our hope that Deuce will help democratize the use of STMs among developers, and that its open-source nature will encourage them to extend the infrastructure with novel features, as has been the case, for instance, with the Jikes RVM [7].

The second part of this thesis (Chapter 3) presents a study of STM-specific compiler optimizations. Each optimization is described, implemented and tested on a mixed testbed of application and micro-benchmarks. Not all STM designs can benefit equally from all of our optimizations. For example, STMs that employ in-place updates, rather than lazy updates, will see less benefit from the optimization that removes redundant memory reads. This is because in a lazy update STM, reading from a memory location entails looking up its most updated value in the write-set, as opposed to an in-place STM where the most updated value is always stored at the memory location itself. We therefore restrict our focus to the Transactional Locking II (TL2) [22] protocol, a lazy update STM which benefits from all the optimizations.

This part makes the following contributions: First we implement a set of common STM-specific analyses and optimizations. Then, we present and implement a set of new analyses and optimizations to reduce overhead of STM instrumentation. And last, we measure and show that our suggested optimizations can achieve significant performance improvements — up to 38% faster, and up to 46-53% more throughput in some workloads.

1.3 Non-Linearizable Objects

Although we strongly believe that TMs can simplify concurrent programming significantly and provide solution for non-experts to utilize multicore, it might take almost a decade until it becomes common practice. Consequently, we expect that the lock-based programming model will remain the main programming method. Despite the fact that the lock-based model gives the programmer a fine-grained way to control the application’s critical sections, it is not uncommon that the outcome exhibits poor scalability.

The source of these phenomena is usually the strict consistency condition imposed by the commonly accepted Linearizability [42]. Linearizability is a useful and intuitive consistency correctness condition that is widely used to reason and prove common
data structures implementations. Intuitively, it requires each run to be equivalent in some sense to a serial run of the algorithm. This equivalence to some serial run imposes strong synchronization requirements that in many cases result in limited scalability and synchronization bottlenecks. In order to overcome this limitation, more relaxed consistency conditions have been introduced. Such alternative consistency conditions for concurrency programming include Sequential consistency\cite{52}, Quiescent consistency\cite{11}, Causal consistency\cite{6}, Release consistency\cite{29}, Eventual consistency\cite{84}, and Timed consistency\cite{81}. But, the semantics of these relaxed conditions is less intuitive and the results are usually unexpected from a layman’s point of view, which makes it an experts only developing tool.

1.3.1 Producer-Consumer Pools

We first start by investigating a common data structure known as producer-consumer pools\cite{40}. That is, collections of unordered objects or tasks. These are a fundamental element of modern multiprocessor software and a target of extensive research and development. Pools show up in many places in concurrent systems. For example, in many applications, one or more \textit{Producer} threads produce items to be consumed by one or more \textit{Consumer} threads \cite{44,87}. These items can be jobs to perform, keystrokes to interpret, purchase orders to execute, or packets to decode. Another type of widely used pool is “resource pools” \cite{83}, in which allocated items are not destroyed but recycled after use in order to be used again.

Pools are accessed by two types of entities: \textit{Producers} and \textit{Consumers}, and offer two main types of operations for example: \textit{Insert} (performed by \textit{Producers}) and \textit{Remove} (performed by \textit{Consumers}). These operations can obey different semantics: can be blocking or non-blocking, depending on the type of the pool definition. When \textit{Insert} is non-blocking, a \textit{Producer} puts its object in the pool and then leaves, whereas when \textit{Insert} is blocking, \textit{Producer} is obliged to block and wait until some \textit{Consumer} removes its object, and only then is free to leave. Similarly, when \textit{Remove} is not blocking, \textit{Consumer} collects an object and leaves, returning a null value in case the pool is empty, whereas blocking \textit{Consumer} will block and wait till an object becomes available.

There are several ways to implement such pools. In the Java\textsuperscript{TM} JDK6.0, for example they are implemented using queues: the \textit{SynchronousQueue}, the \textit{LinkedBlockingQueue}, and the \textit{ConcurrentLinkedQueue}. The synchronous queue provides a “pairing up”
function without buffering; it is entirely symmetric: *Producers* and *Consumers* wait for one another, rendezvous, and leave in pairs, i.e., both *Inserts* and *Removes* are defined as blocking. The other queues provide a buffering mechanism and allow threads to sleep while waiting for their requests to be fulfilled. LinkedBlocking queue implements blocking *Consumers* and non-blocking *Producers*, while in ConcurrentLinkedQueue both *Producers* and consumers are non-blocking. Unfortunately, all these pools, including the new scalable synchronous queues of Lea, Scott, and Shearer [47], are based on centralized structures like a lock-free queue or a stack, and thus are limited in their scalability: the head of the stack or queue is a sequential bottleneck and source of contention. In more detail, queues and stacks enforce LIFO or FIFO semantics, each operation on such a data structure must pass through its head and ”register” its request to obey a certain order. Thus, when used in a multi-threaded environment, threads that access the data structure in parallel create contention on the head, while each awaits its turn to access it.

Pools, on the other hand, by definition, do not have to obey particular order semantics, and items can be inserted and removed any possible order. This thesis shows how to overcome previously mentioned limitations by devising highly-distributed pools based on an *ED-Tree*, a combined variant of the diffracting-tree structure of Shavit and Zemach [76] and the elimination-tree structure of Shavit and Touitou [73]. The ED-Tree does not have a central place through which all threads pass, and thus enables both parallelism and reduced contention. As we explain in Chapter 4.2, an ED-Tree uses randomization to distribute concurrent thread requests to multiple locations so that they collide with one another and can exchange values. It has a specific combinatorial structure called a counting tree [11, 76], which enables requests to be properly distributed even if successful exchanges did not occur.

We use a diffracting tree as the basic construction of the ED-tree, adding queues at the leaves of the tree so that requests either match up or end up properly distributed on the queues at the tree leaves. By “properly distributed” we mean that requests that are not eliminated always end up in the queues: the collection of all the queues together behaves like one large queue. Because the nodes of the tree form a bottleneck if one uses the naïve implementation of diffracting tree, we replace them with highly distributed nodes that use elimination and diffraction on randomly chosen array locations.

The elimination and diffraction tree structures were each proposed respectively
by Shavit and Touiti and Shavit and Zemach [73, 76] and claimed to be effective through simulation [43]. A single level of an elimination array was also used by Hendler, Shavit, and Yerushalmi in implementing shared concurrent stacks and queues [35, 61]. However, elimination trees and diffracting trees were never used to implement real world structures. This is mostly due the fact that there was no need for them: machines with a sufficient level of concurrency and low enough interconnect latency to benefit from them did not exist. Today, multicore machines present the necessary combination of high levels of parallelism and low interconnection costs. Indeed, this work is the first to show that ED-Tree based implementations of data structures from the java.util.concurrent scale impressively on a real machine (a Sun Maramba multicore machine with 2x8 cores and 128 hardware threads), delivering throughput that at high concurrency levels is 10 times that of the existing JDK6.0 algorithms.

But what about low concurrency levels? In their paper describing the JDK6.0 synchronous queue, Lea, Scott, and Shearer [47] suggest that using elimination techniques might indeed benefit the design of synchronous queues at high loads. However, they wonder whether the benefits of reduced contention, achievable by using elimination under high loads, can be made to work at lower levels of concurrency because of the possibility of threads not meeting in the array locations.

This thesis shows that elimination and diffraction techniques can be combined to work well at both high and low loads. There are two main techniques that our ED-Tree implementation uses to make this happen. The first is to have each thread adaptively choose an exponentially varying array range from which it randomly picks a location (backoff in space), and the duration it will wait for another thread at that location. This means that, without coordination, threads will tend to map into a smaller array range as the load decreases, thus increasing chances of a collision. The second component is the introduction of diffraction for colliding threads that are not eliminated because they are performing the same type of operation. The diffraction mechanism enables threads to continue down the tree at a low cost. By the described techniques, we upgrade our algorithm to adapt itself to the current system state, whether it suggests high or low load. The end result is an ED-Tree structure, that, as our empirical testing shows, performs well at both high and low concurrency levels.
1.3. NON-LINEARIZABLE OBJECTS

1.3.2 Quasi-Linearizability

The ED-Tree encouraging results provided us with an incentive to build a more general model. This led us to offer a relaxed version of linearizability that preserves some of the intuition, provides a flexible way to control the level of relaxation, and supports the implementation of more concurrent and scalable data structures.

While Linearizability is the most intuitive condition (providing a serial perception), trying to maintain such strict definition often results in high contention bottlenecks. Despite these limitations, Linearizability is today the most popular consistency model for high-level data structures such as search, queues, and stacks. The definition of Linearizability is equivalent to the following:

- All function calls have a linearization point at some instant between their invocation and response.
- All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition.

This definition is intuitive and composable, making it useful in the implementations of concurrent linearizable data structures from other linearizable objects. But, when using this model to write concurrent highly scalable data structures, the results are often less than satisfactory in terms of performance and scalability.

For example, SEDA[87], the motivating and initiating reason for the current research, is a common design pattern for highly-concurrent servers, which relies heavily on thread pools. Such thread pools are composed of two elements: (i) a set of threads ready to serve tasks and, (ii) a task queue from which the threads consume their tasks. For the task queue, the state-of-the-art concurrent queue of Michael and Scott[58] is usually used. It is based on the fact that enqueue and dequeue might happen concurrently, while threads trying to enqueue should race. Meaning, such a queue, which is not part of the server logic in a highly concurrent system, can in itself become a bottleneck limiting the overall SEDA system utilization. One can claim however, that more than often a thread pool does not need a strict FIFO queue, but rather, what is required is a queue with relaxed linearizability, i.e., that does not allow one task to starve, meaning that it is bypassed by more than a certain number of tasks.

Another common pattern is the shared counter, which in many applications might in itself become a bottleneck. In order to trim down this contention point Aspnes et
al.\cite{11} offered a counting network that reduces contention while maintaining a relaxed consistency condition called quiescent consistency. Such a relaxed counter can be used for example as an ID generator, where the output of this algorithm is a unique ID for each requesting thread, while a strict order is not required. This counter can also match other design patterns, for example a “Statistical Counter.” Modern servers expose many statistical counters, mainly for administration and monitoring. These counters count “online” every operation done on the server. Due to their run-time nature these counters also might easily become a contention point. However, sometimes there is no real need for accurate numbers, rather, a need to capture the general trend. On the other hand the main drawback of the counting network algorithm is also its relaxed consistency. Such relaxation does not provide any upper bound for the “inconsistency.” We show in Section 5.6 that the upper bound is $N \times W$ where $N$ is the number of working threads, and $W$ is the width of the counting network.

Two more common examples for widely used data structures are the Hash Table, and the Stack. While there is a broad range of highly concurrent implementations for a Hash Table as with the former examples the need for a linearizable implementation is often too strict. A very common use case for a Hash Table is a Web Cache. In this case a cache miss while the data is in the Cache might not be a desirable behavior, but can be sacrificed for better scalability. Moreover, even getting stale data for a short while might not be a problem. A similar behavior commonly happens with a Stack. A linearizable LIFO implementation can ordinarily be replaced with an almost LIFO implementation for better scalability. In such an implementation, a pop operation might return an element that is “close” to the top of the stack, but not necessarily the top element. In some uses, this weaker constraint might be acceptable for the gain of better scalability. For example, consider a special form of resource pool that prefers to reuse the most recently-used resource, due to the somewhat higher cost of initializing a resource that was not used recently. Such a resource pool can use a stack to hold its resources and always take the resource from the top, or if replaced with the stack implementation mentioned above, it can take a resource that was used recently but not necessarily the last used resource.

The above examples have motivated us to provide a quantitative definition of the limited non-determinism that the application requirements might allow. We define a consistency condition that is a relaxed Linearizability condition with an upper bound
on the non-determinism. Each operation must be linearizable at most at some bounded distance from its strict linearization point. For example, tasks can be dequeued from a queue not in strict FIFO order but very close to it. That is, there is a linearization order in which every task $t$ that is in the queue only be dequeued if there is no other task $t'$ that is still in the queue, such that $t'$ was enqueued before $t$, and that after $t'$ was enqueued, $k$ or more other tasks were enqueued prior to $t$. This non-strict FIFO order generates the desired behavior in which we can execute any of the first $k$ tasks in the queue, but we are not allowed to bypass the task at the front of the queue more than $k$ times. Our definition is strong and flexible enough to define at the same time (continuing the above example) that a dequeue that returns empty may not be reordered, i.e., it has to be in its strict linearizable order. In this thesis we introduce a formal definition of the quasi-linearizability condition that captures this condition. This condition introduces some degree of non-determinism, but is useful to prove the quasi-linearizability of different implementations as exemplified in later sections.

1.3.2.1 Related Work

Many models were offered as weaker alternatives to Linearizability. Three of them are Quiescent consistency[11], Eventual consistency[84] and Timed consistency[81].

Quiescent consistency model provides high-performance at the expense of weaker constraints satisfied by the system. An object is in a quiescent state if currently there is no pending or executing operation on that object. And the model assure that operations whose occurrence is separated by a quiescent state should appear in the order of their occurrence and may not be reordered. Eventual consistency is a specific form of weak consistency; e.g, a storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. These two models, in most cases, allow better concurrency but on the other hand do not provide any strict upper bound or an adaptive way to determine the “inconsistency” gap when compared to Linearizability. The Timed consistency does provide a special form of upper bound on the “inconsistency.” This model adds the notion of time to the occurrences of events and not just order, roughly speaking, timed consistency models require that if a write operation is executed at time $t$, it must be visible to all processes by time $t + \Delta$. As a result the actual implementations and guarantees they provide that satisfy either Timed consistency or quasi-linearizable conditions are very different.
1.4 Outline

The rest of the dissertation is organized as follows. We first describe Deuce in Chapter 2. We start in Section 2.1 from the perspective of the developer of a concurrent application. Then we move to discuss the implementation of the framework in Section 2.2. Section 2.3 shows how it can be extended, by the means of the STM back ends. Finally, in Section 2.4, we evaluate its performance.

In Chapter 3 we review our extension of Deuce to support STM-specific compiler optimizations. First, we start in Section 3.1 with a short overview, surveying the field of STM-specific compiler optimizations and describing the methodology of our work. Sections 3.2 and 3.3 describe the STM compiler optimizations. In Section 3.2 we describe common STM compiler optimizations. Section 3.3 presents our new STM compiler optimizations, the main contribution of this chapter, respectively. Last, Section 3.4 presents performance evaluation of the different optimizations.

Chapter 4 introduces a novel implementation for distributed pools based on an ED-Tree. We start with an overview in Section 4.1 on Diffracting Tress and Elimination. Next, in Section 4.2 we provide a glance at the ED-Tree algorithm. This chapter concludes with a deep dive presentation of the algorithm implementation (Section 4.3) followed by an evaluation of the algorithm (Section 4.4).

The last part of this thesis presented in Chapter 5, provides a formal definition of a new model call Quasi-Linearizability. This chapter begins with an overview, Section 5.1) covers the various consistency models in general, including Linearizability. Next, in Section 5.2, we provide a formal definition of Quasi-Linearizability and treat the abnormalities that might arise from the definition. Section 5.3 demonstrates two implementations of a FIFO queue that utilize this definition, followed by an empirical evaluation in Section 5.5. Finally, in Section 5.6 we show that this definition is implicitly used in existing algorithms by showing that a Bitonic[W] Counting Network[11] is in fact Quasi-Linearizable.
Chapter 2

Noninvasive Concurrency with a Java™ STM
2.1 Overview

One of the main goals in designing the Deuce API was to keep it simple. A survey of the past designs (see previous section) reveals three main approaches: (i) adding a new reserved keyword to mark a block as atomic, e.g., atomic in AtomJava [45]; (ii) using explicit method calls to demarcate transactions and access shared objects, e.g., DSTM2 [37] and CCR [33]; or (iii) requiring atomic classes to implement a predefined interface or extend a base class, and to implement a set of methods DSTM [38]. The first approach requires modifying the compiler and/or the JVM, while the others are intrusive from the programmer’s perspective as they require significant modifications to the code (even though some systems use semi-automatic weaving mechanisms such as aspect-oriented programming to ease the task of the programmer, for example, LSASTM [69]).

In contrast, Deuce has been designed to avoid any addition to the language or changes to the JVM. In particular, no special code is necessary to indicate which objects are transactional, no method needs to be implemented to supporting transactional objects (e.g., cloning the state as in DSTM [38]). This allows Deuce to seamlessly support transactional accesses to pre-compiled libraries. The only piece of information that must be specified by the programmer is, obviously, which part of the code should execute atomically in the context of a transaction.

To that end, Deuce relies on Java™ annotations as shown in Figure 2.1. Introduced as a new feature in Java™ 5, annotations allow programmers to mark a method with metadata that can be consulted at class loading time. Deuce introduces new types of annotations to mark methods as atomic: their execution will take place in the context of a transaction.

```java
@Atomic public int sum(List list) {
    int total = 0;
    for (Node n : list)
        total += n.getValue();
    return total;
}
```

Figure 2.1: @atomic method example.

This approach has several advantages, both technically and semantically. First technically, the smallest code block Java™ can annotate is a method, which simplifies
the instrumentation process of DEUCE and provides a simple model for the programmer. Second, atomic annotations operate at the same level as synchronized methods [54], which execute in a mutually exclusion manner on a given object; therefore, atomic methods provide a familiar programming model, as shown in Figure 2.2.

```java
synchronized public int sum(List list) {
    int total = 0;
    for (Node n : list)
        total += n.getValue();
    return total;
}
```

Figure 2.2: synchronized method example.

From a semantic point of view, implementing atomic blocks at the granularity of methods removes the need to deal with local variables as part of the transaction. In particular, since Java™ doesn’t allow any access to stack variables outside the current method, the STM can avoid logging many memory accesses. For instance, in Figure 2.3, a finer-granularity atomic block would require costly logging of the total local variable (otherwise the method would yield incorrect results upon abort) whereas no logging would be necessary when considering atomic blocks at the granularity of individual methods. In cases when finer granularity is desirable, DEUCE can be used in combination with the TMJAVAX [31] pre-compiler as shown in Figure 2.3.

```java
public int sum(List list) {
    int total = 0;
    atomic {
        for (Node n : list)
            total += n.getValue();
    }
    return total;
}
```

Figure 2.3: Atomic block example.

### 2.1.1 Concurrent Programming with DEUCE

To illustrate the use and implementation of DEUCE, we will consider a well-known but non-trivial data structure: the skip list [67]. A skip list is a probabilistic structure based on multiple parallel, sorted linked lists, with efficient search time in $\mathcal{O}(\log n)$. 

```java
public int sum(List list) {
    int total = 0;
    atomic {
        for (Node n : list)
            total += n.getValue();
    }
    return total;
}
```

Figure 2.3: Atomic block example.
Figure 2.4 shows a partial implementation of skip list, with an inner class representing nodes and a method to search for a value through the list. The key observation in this code is that transactifying an application is as easy as adding @Atomic annotations to methods that should execute as transactions. No code needs to be changed within the method or elsewhere in the class. Interestingly, the linked list directly manipulates arrays and accesses public fields from outside their enclosing objects, which would not be possible with DSTM2 or LSA-STM.

```java
public class SkipList {
    // Inner class - represents a node in the SkipList
    private static class Node {
        public final int value;
        public final Node[] forward;
        // ...
        public Node(int level, int v) {
            value = v;
            forward = new Node[level + 1];
        }
        // ...
    }
    private static int MAX_LEVEL = 32;
    private int level;
    private final Node head;

    @Atomic(retries=64)
    public boolean contains(int v) {
        Node node = head;
        for (int i = level; i >= 0; i--) {
            Node next = node.forward[i];
            while (next.value < v) {
                node = next;
                next = node.forward[i];
            }
        }
        node = node.forward[0];
        return (node.value == v);
    }

    @Atomic
    public boolean add(int value) {
        // ...
        return true;
    }
```

Figure 2.4: @Atomic method example.
One can also observe that the @Atomic annotation provides one configurable attribute, retries, to optionally limit the amount of retries the transaction attempts (at most 64 times in the example - Line 17). A TransactionException is thrown in case this limit is reached. Alternatively one can envision providing timeout instead (Which we might add in future versions).

2.1.2 The DEUCE Runtime

A DEUCE application is compiled with a regular Java™ compiler. Upon execution, one needs to specify a Java™ agent that allows DEUCE to intercept every class loaded and manipulate it before it is loaded by the JVM. The agent is simply specified on the command line as a parameter to the JVM, as follows:

```
java -javaagent:deuceAgent.jar MainClass args...
```

As will be discussed in the next section, DEUCE instruments every class that may be used from within a transaction, not only classes that have @Atomic annotations. If it is known that a class will never be used in the context of a transaction, one can prevent it from being instrumented by annotating the class with @Exclude or providing exclusion lists to the DEUCE agent. This will speed up the application loading time yet should not affect execution speed.

2.2 The DEUCE Implementation

This section describes the implementation of the DEUCE framework. We first give a high-level overview of its main components. Then, we explain the process of code instrumentation. Finally, we describe various optimizations that enhance the performance of transactional code.

2.2.1 Overview

The DEUCE framework is conceptually made up of three layers, as shown in Figure 2.5

(I) The application layer, which consists of user classes written without any relationship to the STM implementation, except for annotations added to atomic methods.
CHAPTER 2. NONINVASIVE CONCURRENCY WITH A JAVA™ STM

Figure 2.5: Main components of the Deuce architecture.

(II) The Deuce runtime layer, which orchestrates the interactions between transactions executed by the application and the underlying STM implementation.

(III) The layer of actual STM libraries that implement the Deuce context API (see Section 2.3), including a single-global-lock implementation (denoted as “Lock” in the figure, and used as a performance “reference point” for all other libraries).

In addition, the Deuce agent intercepts classes as they are loaded and instruments them before they start executing.

2.2.2 Instrumentation Framework

Deuce’s instrumentation engine is based on ASM [14], an all-purpose Java™ bytecode manipulation and analysis framework. It can be used to modify existing classes, or to dynamically generate classes, directly in binary form. The Deuce Java™ agent uses ASM to dynamically instrument classes as they are loaded, before their first instance is created in the virtual machine. During instrumentation, new fields and methods are added, and the code of transactional methods is transformed. We now describe the different manipulations performed by the Java™ agent.

2.2.2.1 Fields

For each instance field in any loaded class, Deuce adds a synthetic constant field (final static public) that represents the relative position of the field in the class. This value, together with the instance of the class, uniquely identifies a field, and is used by the STM implementation to log field accesses.

Static fields in Java™ are effectively fields of the enclosing class. To designate static fields, Deuce defines for each class a constant that represents the base class, and
2.2. THE DEUCES IMPLEMENTATION

```java
public class SkipList {
    public static final long __CLASS_BASE__ = ...;
    public static final long MAX_LEVEL_address__ = ...;
    public static final long level_address__ = ...;
    // ...
    private static int MAX_LEVEL = 32;
    private int level;
    // ...
}
```

Figure 2.6: Fields address.

can be used in combination with the field position instead of the class instance.

For instance, in Figure 2.6, the `level` field is represented by `level_address__` while the `SkipList` base class is represented by `__CLASS_BASE__`.

2.2.2.2 Accessors

For each field of any loaded class, Deuces adds synthetic static accessors used to trigger field’s access events on the local context.

```java
    public class SkipList {
        private int level;
        // ...
        // Synthetic getter
        public int level__Getter$ (Context c) {
            c.beforeReadAccess(this, level_address__);
            return c.onReadAccess(this, level, level_address__);
        }
        // Synthetic setter
        public void level__Setter$ (int v, Context c) {
            c.onWriteAccess(this, v, level_address__);
        }
        // ...
    }
```

Figure 2.7: Fields accessors.

Figure 2.7 shows the synthetic accessors of class `SkipList`. The getter `level__Getter$` receives the current field value and a context, and triggers two events: `beforeReadAccess` and `onReadAccess` (the result of the latter is returned by the getter). The setter receives a context and yields a single event: `onWriteAccess`. The reason for having two events in the getter is technical: the “before” and “after” events...
allow the STM backend to verify that the value of the field, which is accessed between both events without using costly reflection mechanisms, is consistent. This is typically the case in time-based STM algorithms like TL2 and LSA that ship with Deuce.

```java
private static class Node {
    public Node[] forward;
    // ...}

    // Original method
    public void setForward(int level, Node next) {
        forward[level] = next;
    }

    // Synthetic duplicate method
    public void setForward(int level, Node next, Context c) {
        Node[] f = forward.Getter$(c)
        forward.Setter$(f, level, next, c)
    }
    // ...}

Figure 2.8: Duplicate method.

2.2.2.3 Duplicate methods

In order to avoid any performance penalty for non transactional code, and since Deuce provides weak atomicity, Deuce duplicates each method to provide two distinct versions. The first version is identical to the original method: it does not trigger an event upon memory access and, consequently, does not impose any transactional overhead. The second version is a synthetic method with an extra Context parameter. In this instrumented copy of the original method, all field accesses (except for final fields) are replaced by calls to the associated transactional accessors. Figure 2.6 shows the two versions of a method of class Node. The second synthetic overload replaces forward[level] = next by calls to the synthetic getter and setter of the forward array. The first call (Line 12) obtains the reference to the array object, while the second one (Line 13) changes the specified array element (note that getters and setters for array elements have an extra index parameter).
2.2. THE DEUCE IMPLEMENTATION

```java
public class SkipList {
    // ...

    // Original method instrumented
    public boolean contains(int v) {
        Throwable throwable = null;
        Context context = ContextDelegator.getInstance();
        boolean commit = true;
        boolean result;

        for (int i = 64; i > 0; --i) {
            context.init();
            try {
                result = contains(v, context);
            } catch (TransactionException ex) {
                // Must rollback
                commit = false;
            } catch (Throwable ex) {
                throwable = ex;
            }
            // Try to commit
            if (commit) {
                if (context.commit()) {
                    if (throwable == null)
                        return result;
                // Rethrow application exception
                throw (IOException) throwable;
                }
            } else {
                context.rollback();
                commit = true;
            }
        } // Retry loop
        throw new TransactionException();
    }

    // Synthetic duplicate method
    public boolean contains(int v, Context c) {
        Node node = head__Getter$(c);
        // ...
    }

    Figure 2.9: Atomic method.
```
2.2.2.4 Atomic methods

The duplication process described above has one exception: a method annotated as @Atomic does not need the first uninstrumented version. Instead, the original method is replaced by a transactional version that calls the instrumented version from within a transaction that executes in a loop. The process repeats as long as the transaction aborts and a bound on the number of allowed retries is not reached. Figure 2.9 shows the transactional version of method contains where the original contains method was replaced by a 64 retry loop (as set in the retries attribute). Each loop calls the duplicated method (Line 15) with the context and unless a TransactionException was thrown (Line 18) tries to commit the transaction (Line 25). Otherwise the a rollback is called and another retry starts (Line 32).

2.2.3 Summary

To summarize, Deuce performs the following instrumentation operations on the Java™ bytecode at load-time.

- For every field, a constant is added to keep the relative location of the field in the class for fast access.
- For every field, two accessors are added (getter and setter).
- For every class, a constant is added to keep the reference to the class definition and allow fast access to static fields.
- Every (non-@Atomic) method is duplicated to provide an atomic and a non-atomic version.
- For every @Atomic method, an atomic version is created and the original version is replaced with a retry loop calling the atomic version in the context of a new transaction.

2.2.4 Optimizations

During the instrumentation, we perform several optimizations to improve the performance of Deuce. First, we do not instrument accesses to final fields as they cannot be modified after creation. This optimization, together with the declaration of final fields whenever possible in the application code, dramatically reduces the overhead.
Second, fields accessed as part of the constructor are ignored as they are not accessible by concurrent threads until the constructor returns.

Third, instead of generating accessor methods, DEUCE actually inlines the code of the getters and setters directly in the transactional code. We have observed a slight performance improvement from this optimization.

Fourth, we chose to use the sun.misc.Unsafe pseudo-standard internal library to implement fast reflection, as it proved to be vastly more efficient than the standard Java\textsuperscript{TM} reflection mechanisms. The implementation using sun.misc.Unsafe even outperformed the approach taken in AtomJava \cite{45}, which is based on using an anonymous class per field to replace reflection.

Finally, we tried to limit as much as possible the stress on the garbage collector, notably by using object pools when keeping track of accessed fields (read and write sets) in threads. In order to avoid any contention on the pool, we had each thread keep a separate object pool as part of its context.

Together, the above optimizations helped to significantly decrease the implementation overhead, in some of our benchmarks this improved performance by almost an order of magnitude (i.e., tenfold faster) as compared to our initial implementation. Moreover, these optimizations motivated us and triggered the static analysis optimizations described in Chapter 3.

\section{2.3 Customizing Concurrency Control}

DEUCE was designed to provide a research platform for STM algorithms. In order to provide a simple API for researchers to plug in their own STM algorithm’s implementation, DEUCE defines the Context API as shown in Figure 2.10. The API includes an init method (Line 2), called once before the transaction starts and then upon each retry, allowing the transaction to initialize its internal data structures. The atomicBlockId argument allows the transaction to log information about the specific atomic block (statically assigned in the bytecode).

One of the heuristics we added to the LSA implementation is, following TL2 \cite{22}, that each one of the atomic blocks will initially be a read-only block. It will be converted to become a writable block upon retry, once it encounters a first write. Using this method, read-only blocks can save most of the overhead of logging the fields’ access.
Another heuristic is that `commit` is called in case the atomic block finishes without a `TransactionException` and can return `false` in case the commit fails, which in turn will cause a retry. A `rollback` is called when a `TransactionException` is thrown during the atomic block (this can be used by the business logic to force a retry).

The rest of the methods are called upon field access: a field read event will trigger a `beforeReadAccess` event followed by a `onReadAccess` event. A write field event will trigger an `onWriteAccess` event. DEUCE currently includes three `Context` implementations, `tl2.Context`, `lsa.Context`, and `norec.Context`, implementing the TL2 [22], LSA [69], and NOrec [18] STM algorithms respectively. DEUCE also provides a reference implementation based on a single global lock. Since a global lock doesn’t log any field access, it doesn’t implement the Context interface and doesn’t impose any overhead on the fields’ access.

### 2.3.1 TL2 Context

The TL2 context is a straightforward implementation of the TL2 algorithm [22]. The general principle is as follows.

TL2 is a lazy-update STM, so values are only written to memory at commit time; therefore locks are held for a very short amount of time. TL2 uses a shared array of revokable versioned locks, with each object field being mapped to a single lock. Each
lock has a version that corresponds to the commit timestamp of the transaction that last updated some field protected by this lock. Timestamps are acquired upon commit from a global time base, implemented as a simple counter, updated as infrequently as possible by writing transactions.

Upon write, the transaction buffers the update in its write-set, a thread-local data structure holding pairs of memory locations and last written values. When reading some data, a transaction first checks to see if the read memory address already appears in the write-set, and if so, its written value is returned. Otherwise, a transaction pre-validation step checks that the associated timestamp is valid, in other words, not more recent than the time when the transaction started. If valid, it keeps track of the accessed location in its read set, another thread-local data structure, holding memory locations. The read is followed by a post-validation step, which checks that the associated version is still the same as it was in the pre-validation step.

At commit time, the transaction acquires (using an atomic compare-and-set operation) the locks protecting all written fields, verifies that all entries in the read set are still valid, acquires a unique commit timestamp, writes the modified fields to memory, and releases the locks. If locks cannot be acquired or validation fails, the transaction aborts, discarding buffered updates. If the validation passes, the global clock is incremented, the items in

DEUCE’s version of TL2 supports weak isolation \[57\]; this means that a program accessing the same memory location both transactionally and non-transactionally may encounter unexpected behavior. In a weakly-isolated STM, memory locations that are accessed from within transactions, should not be concurrently accessed from without transactions.

In addition, DEUCE’s TL2 version supports flat nesting \[5\]; this means that a transaction B, nested inside a transaction A, shared its read-set and write-set with transaction A. B’s changes are visible to other threads only after A commits. If B aborts, so does A. The effect is that transactions A and B form a single, flat transaction, hence the name.

2.3.2 LSA Context

The LSA context uses the LSA algorithm \[69\] that was developed concurrently with TL2 and is based on a similar design. The main differences are that (1) LSA acquires
locks as fields are written (encounter order), instead of at commit time, and (2) it performs “incremental validation” to avoid aborting transactions that read data that has been modified more recently than the start time of the transaction.

Both TL2 and LSA take a simple approach to conflict management: they simply abort and restart (possibly after a delay) the transaction that encounters the conflict. This strategy is simple and efficient to implement, because transactions unilaterally an abort without any synchronization with others. However, this can sometimes hamper progress by producing livelocks. Deuce also supports variants of the TL2 and LSA algorithms that provide modular contention management strategies (as first proposed in [38]), at the price of some additional synchronization overhead at runtime.

2.3.3 NOrec Context

The NOrec context implements the lightweight algorithm proposed in [18]. Roughly speaking, NOrec differs from TL2 and LSA in that it uses a single ownership record (a global versioned lock) and relies on value-based validation in addition to time-based validation. The rationale of this design is to avoid having to pay the runtime overheads associated with accessing multiple ownership records while still providing a reasonable level of concurrency thanks value-based validation which helps identify false conflicts. This algorithm was shown to be efficient at low thread counts, but tends to fall apart as concurrency increases. In contrast, TL2 and LSA provide better scalability, performing better as concurrent threads are added or when there are more frequent but disjoint commits of software transactions.

2.4 Performance Evaluation

We evaluated the performance of Deuce on a Sun UltraSPARC™ T2 Plus multicore machine. This machine has two chips, each with 8 cores running at 1.2 GHz, each core with 8 hardware threads, so 64 way parallelism on a processor and 128 way parallelism across the machine. There is obviously a higher latency when going to memory across the machine (a two fold slowdown). We also evaluated on Azul Vega2™ This machine has two chips, each with 48 cores.
2.4. PERFORMANCE EVALUATION

2.4.1 Overhead

We first briefly discuss the overheads introduced by the DEUCE agent when instrumenting Java\textsuperscript{TM} classes. Table 2.1 shows the memory footprint and the processing overhead when processing compiled libraries: rt.jar is the runtime library containing all Java\textsuperscript{TM} built-in classes for JDK 6; cache4j.jar (version 0.4) is a widely used in-memory cache for Java\textsuperscript{TM} objects; JavaGrande (version 1.0) is a well-known collection of Java\textsuperscript{TM} benchmarks. Instrumentation times were measured on an Intel Core 2 Duo\textsuperscript{TM} CPU running at 1.8 GHz. Note that instrumentation was executed serially, that is to say, without exploiting multiple cores.

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory</th>
<th>Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Instrumented</td>
</tr>
<tr>
<td>rt.jar</td>
<td>50 MB</td>
<td>115 MB</td>
</tr>
<tr>
<td>cache4j.jar</td>
<td>248 KB</td>
<td>473 KB</td>
</tr>
<tr>
<td>JavaGrande</td>
<td>360 KB</td>
<td>679 KB</td>
</tr>
</tbody>
</table>

Table 2.1: Memory footprint and processing overhead

As expected, the size of the code approximately doubles because DEUCE duplicates each method, and the processing overhead is proportional to the size of the code. However, because Java\textsuperscript{TM} uses lazy class loading, only the classes that are actually used in an application will be instrumented at load time. Therefore, the overheads of DEUCE are negligible considering individual classes are loaded on demand.

In terms of execution time, instrumented classes, that are not invoked within a transaction, incur no performance penalty as the original method executes.

2.4.2 Benchmarks

We tested the four built-in DEUCE STM options: LSA, TL2, NOrec, and the simple global lock. Our LSA version captures many of the properties of the LSA-STM \cite{70} framework. The TL2 \cite{22} form provides an alternative STM implementation that acquires locks at commit time using a write set, as opposed to the encounter order locking approach used in LSA.

Our experiments included three classical micro-benchmarks: a red/black tree, a sorted linked list, and a skip list, on which we performed concurrent insert, delete, and search operations. We controlled the size of the data structures, which remained
constant for the duration of the experiments, and the mix of operations.

We also experimented with three real-world applications, the first implements the Lee routing algorithm (as presented in Lee-TM [10]). It is worth noting that adapting the application to DEUCE was straightforward, with little more than a change of synchronized blocks into atomic methods. The second application is called Cache4J, an LRU cache implementation. Again adapting the application to DEUCE was straightforward. The third application comes from the widely used STAMP [59] TM benchmark suite.

2.4.2.1 Micro-benchmarks

We began by comparing DEUCE to DSTM2, the only competing STM framework that does not require compiler support. We benchmarked DSTM2 and DEUCE based on the Red/Black tree benchmark provided with the DSTM2 framework. We tried many variations of operation combinations and levels of concurrency on both the Azul and Sun machines. Comparison results for a tree with 10k elements are shown in Figure 2.11, we found that in all the benchmarks DSTM2 was about 100 times (two orders of magnitude) slower than DEUCE. Our results are consistent with those published in [37], where DSTM2’s throughput, measured in operations per second, is in the thousands, while DEUCE’s throughput is in the millions. Based on these experiments, we believe one can conclude that DEUCE is the first viable compiler and JVM independent Java™ STM framework.

On the other hand we observe that the DEUCE STMs implementations scale in an impressive way even with 20% updates (up to 32 threads). While DSTM2 shows no scalability. When we investigated deeper we found out that most of DSTM2 overhead rest in two areas. The most significant area is the contention manager which acts as a contention point, the second area is the reflection mechanism used by DSTM2 to
retrieve and assign values during the transaction and in commit.

Next we present the results of the tests with the linked list and the skip list on the Sun and Azul machines. The linked list benchmark is known to produce a large degree of contention as all threads must traverse the same long sequence of nodes to reach the target element. Results of experiments are shown in Figure 2.12 and Figure 2.13 for a linked list with 256 elements and different update rates. We observe that because there is little potential for parallelism, the single lock, which performs significantly less lock acquisition and tracking work than the STMs, wins in all three benchmarks. The STMs scale slightly until 30 threads when there are up to 5% updates, and then drop due to a rise in overhead with no benefit in parallelism. With 20% updates, the max is reached at about 5 threads, because as concurrency further increases one can expect at least 2 concurrent updates, and the STMs suffer from repeated transactional aborts.

![Figure 2.12: The linked list benchmark (Sun).](image1)

![Figure 2.13: The linked list benchmark (Azul).](image2)

![Figure 2.14: The skiplist benchmark (Sun).](image3)

Finally, we consider results from the skip list benchmark. Skip lists [66] allow
significantly greater potential for parallelism than linked lists because different threads are likely to traverse the data structure following distinct paths. Results for a list with 16k elements are shown in Figure 2.14 and Figure 2.15. We observe that the STMs scale in an impressive way, as long as there is an increase in the level of parallelism, and provide great scalability even with 20% updates as long as the abort rate remains at a reasonable level. We added a benchmark with 50% updates to show that there is a limit to their scalability. When we increase the fraction of updates to 50%, the STMs in Figure 2.15 reach a “melting point” much earlier (at about 10 threads versus 100 threads in the 20% benchmark) and overall the lock wins again because the abort rate is high and the STMs incur and overhead without a gain in parallelism. We note that typically search structures have about 10% updates.

2.4.2.2 Real applications

Our last two benchmarks demonstrate how simple it is to replace the critical sections with transactions, both benchmarks spend almost 100 percent of the time in critical sections. The first takes a serial version of the Lee routing algorithm and demonstrates how a simple replacement of the critical sections by transactions significantly improves scalability. The second takes a non-multi-threaded lock based a LRU cache implementation (Cache4J) and shows that it is straightforward to replace the critical sections, but scalability isn’t promised.

Circuit routing is the process of automatically producing an interconnection between electronic components. Lee’s routing algorithm is an attractive candidate for parallelization since circuits (as shown in [10]) consist of thousands of routes, each of which can potentially be routed concurrently.

The graph in Figure 2.16 shows execution time (that is, latency, not throughput) for three different boards with the TL2, LSA, and NOrec algorithms. As can be seen,
Deuce scales well with all algorithms, with the overall time decreasing even in the case of a large board (MemBoard). NOrec exhibits better performance with a single thread due to the lower overhead of its single ownership record. With more threads, the performance of all algorithms is virtually identical. This can be explained by the natural parallelism and low contention of this benchmark.

Cache4J is an LRU lock-based cache implementation. The implementation is based on two internal data structures, a tree and a hash-map. The tree manages the LRU while the hash-map holds the data. The Cache4J implementation is based on a single global lock which naturally leads to no scalability. The graph in Figure 2.17 shows the result of replacing the single lock with transactions using the LSA algorithm. As can be seen, Deuce doesn’t scale well, with the overall throughput slightly decreasing. A quick profiling shows that the fact that Cache4J is an LRU cache implies that every get operation also updated the internal tree. This alone makes every transaction an update transaction. Yet, the fact that the total throughput remains almost the same even with 80 threads is encouraging due to the fact that transactional memory’s main advantages, besides scalability, are code simplicity and robustness.

Finally, we tested Deuce on the Vacation benchmark from STAMP [59], which is
CHAPTER 2. NONINVASIVE CONCURRENCE WITH A JAVA™ STM

Figure 2.18: The Vacation benchmark from the STAMP suite (Sun).

the most widely used TM benchmark suite. As STAMP was been originally written in C, we used a Java™ port of the original benchmarks adapted for Deuce. The Vacation application models an online travel reservation system, implemented as a set of trees that keep track of customers and their reservations. Threads spend time executing transactions, and the contention level is moderate. Figure 2.18 shows the execution time of Vacation with the TL2, LSA, and NOrec algorithms. We can observe that both TL2 and LSA scale well with the number of cores. In contrast, NOrec shows almost no scalability, although it performs better than TL2 and LSA under low thread counts. This can be explained by the fact that (1) NOrec’s single ownership record is penalized by the size of the transactions, and (2) value-based validation is quite expensive in Java™ since it requires accessing memory via APIs that are slower than the direct memory accesses provided by unmanaged languages.

In conclusion, Deuce shows the typical performance patterns of an STM, good scalability when there is a potential for parallelism, and unimpressive performance when parallelism is low. The results, however, are very encouraging considering the fact that Deuce, a pure Java™ library with all the implied overheads, shows scalability even at low thread counts.

2.5 Conclusion

We introduced Deuce, a novel open-source Java™ framework for transactional memory. As we showed, Deuce is the first efficient fully featured Java™ STM framework that can be added to an existing application without changes to its compiler or libraries. It demonstrates that one can design an efficient, pure Java™ STM without a compiler support, even though language support as provided by the companion front-end TM-
Java provides additional expressiveness.

In conclusion, Table 2.2 shows a general overview comparing the two well-known Java™ STMs AtomJava¹ and DTMS2 vs. DEUCe. This comparison shows that DEUCe ranks highest overall, and provides novel capabilities which yield much better usability and performance than formerly existed.

<table>
<thead>
<tr>
<th>Locks</th>
<th>AtomJava</th>
<th>DSTM2</th>
<th>DEUCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrumentation</td>
<td>Source</td>
<td>Runtime</td>
<td>Runtime</td>
</tr>
<tr>
<td>API</td>
<td>Non-Intrusive</td>
<td>Intrusive</td>
<td>Non-Intrusive</td>
</tr>
<tr>
<td>Libraries support</td>
<td>None</td>
<td>None</td>
<td>Runtime &amp; offline</td>
</tr>
<tr>
<td>Fields access</td>
<td>Anonymous classes</td>
<td>Reflection</td>
<td>Low level unsafe</td>
</tr>
<tr>
<td>Execution overhead</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Extensible</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Transaction context</td>
<td>atomic block</td>
<td>Callable object</td>
<td>@Atomic annotation</td>
</tr>
</tbody>
</table>

Table 2.2: Comparing AtomJava, DSTM2, and DEUCe

¹We did not show comparison benchmark results with AtomJava due to technical limitations and bugs found in its current implementation, but we have observed on small benchmarks that DEUCe outperforms AtomJava by a comfortable margin.
Chapter 3

Lowering STM Overhead with Static Analysis
3.1. Overview

As we marked in the introduction, in the past several years there has been a flurry of software transactional memory design and implementation work; however, with the notable exception of transactional C/C++ compilers [64], many of the STM initiatives have remained academic experiments. There are several reasons for this; major among them is the large performance overhead [17]. In order to make a piece of code transactional, it usually undergoes an instrumentation process which replaces memory accesses with calls to STM library functions. These functions handle the book-keeping of logging, committing, and rolling-back of values according to the STM protocol. Naive instrumentation introduces redundant function calls, for example, for values that are provably immutable. In addition, an STM with homogeneous implementation of its functions, while general and correct, will necessarily be less efficient than a highly-heterogeneous implementation: the latter can offer specialized functions that handle some specific cases more efficiently. For example, a homogeneous STM may offer a general $\text{STMRead}()$ method, while a heterogeneous STM may offer also a specialized $\text{STMReadThreadLocal}()$ method that assumes that the value read is thread-local, and as a consequence, can optimize away validations of that value.

In Chapter 2 we introduced DEUCE which from its initial design introduced couple of trivial optimizations. These optimizations, as showed, improved the STM throughput significantly. But, although DEUCE outperformed any other known implementation it is not uncommon that DEUCE high overhead might cause more harm than good. Such overhead, which is imposed by definition by any STM, can convince any developer to prefer risking working with Locks or abandon parallel programming to the comfort of serial programming. In this part of the thesis we present a study of STM-specific compiler optimizations. Each optimization is described, implemented and tested on a mixed testbed of application and micro-benchmarks.

Not all STM designs can benefit equally well from all of our optimizations. For example, STMs that employ in-place updates, rather than lazy updates, will see less benefit from the optimization that removes redundant memory reads. This is because in a lazy update STM, reading from a memory location entails looking up its most updated value in the write-set, as opposed to an in-place STM where the most updated value is always stored at the memory location itself. We therefore restrict our focus to
the TL2 protocol (as described in Subsection 2.3.1), an in-place STM which benefits from all the optimizations.

3.1.1 Novel New Optimizations

We begin with a high-level description of our novel optimizations.

**Read-set insert elimination on read-only transactions** Transactions which do not modify any shared memory can be exempt from inserting its read memory locations to the read-set. We provide an analysis that discovers such read-only transactions.

**Redundant memory reads elimination** Load elimination, a compiler technique that reduces the amount of memory reads by storing read values in local variables and using these variables instead of reading from memory. This allows us to reduce the number of costly STM library calls.

**Redundant memory writes elimination** Scalar promotion, a compiler technique that avoids redundant stores to memory locations by storing to a local variable. Similar to load elimination, this optimization allows us to reduce the number of costly STM library calls.

**Redundant write-set lookups elimination** Discover memory accesses that read locations which have not been previously written to by the same transaction. Instrumentation for such reads can avoid write-set lookup.

**Redundant write-set record-keeping elimination** Discover memory accesses that write to locations which will not be subsequently read by the same transaction. Instrumentation for such writes can therefore be made cheaper, e.g., by avoiding insertion to a Bloom filter.

**Transaction rescoping** Tighten the atomic block if we discover that statements at the beginning or end of it can be safely hoisted outside of the atomic block.

3.1.2 Known Optimizations

In addition to the new optimizations, we have implemented the following optimizations which have been used in other STMs.
Instrumentation elimination of immutable and transaction-local memory

Such accesses can directly access the memory locations instead of going through
the STM library function call.

Eliminate synchronization for thread-local memory

Thread-local memory can
be read without requiring validation, and can be updated without locking.

3.1.3 Implementing the Analyses

In order to implement and test the STM compiler optimizations, we added optimization
passes that operate before Deuce’s instrumentation phase. The passes collect inform-
ation from the complete application about optimization opportunities, and in some
cases (PRE-related optimizations) transform the code. The information is transferred
to Deuce which uses it when instrumenting the code. The optimization passes are
implemented using Soot [82], a powerful Java™ optimization framework.

Deuce’s STM library is homogenous. In order to allow its methods to take ad-
antage of specific cases where optimization is possible, we enhance each of its STM
functions to accept an extra incoming parameter, advice. This parameter is a simple
bit-set representing information that was pre-calculated in the optimization passes and
may help fine-tune the instrumentation. For example, when writing to a field that
will not be read, the advice passed to the STM write function will have ‘1’ in the bit
corresponding to “no-read-after-write.”

3.2 STM Optimizations - Present Techniques

In this section and the next, we present a number of STM compiler optimizations. Each
optimization is described by:

- **Introduction:** An explanation of the optimization opportunity, usually with the
  help of a code example.

- **Analysis:** A static analysis algorithm that discovers the optimization opportu-
nity.

- **Applicability:** A measure of how effective the optimization is when applied
to the benchmarks. This measure is optimization-specific. For example, in op-
timizations that reduce the amount of synchronization operations, we measure
how many such operations were removed; in optimizations that remove redundant memory reads, we measure how many such reads were removed; and so on.

The performance evaluation of the optimizations is discussed in Section 3.4.

3.2.1 Immutable Memory

3.2.1.1 Introduction

A shared memory location is immutable if different threads always observe the same value when reading it. A common STM compiler optimization is to avoid instrumenting reads of immutable memory. As an example, consider the `sumPair()` method in Figure 3.1. If we can statically deduce that `Pair`’s `x` and `y` fields are immutable, the STM can directly access the two fields instead of replacing access to them by a call to an expensive STM library function.

```java
@Atomic public sumPair(Pair p) {
    return p.x + p.y;
}
```

Figure 3.1: Immutable field access.

Many works (e.g., [1]) consider a field to be immutable if it is declared with the Java™ `final` modifier. This is both dangerous and sub-optimal. It is dangerous because, surprisingly, fields annotated with `final` can still appear to change their value over time. Such a situation happens if an object which is not yet fully initialized is passed around to another thread. The other thread may observe an uninitialized (zero or null) value at first, while later reads of the same field will observe the intended initialized value. Detecting immutability with `final` fields is sub-optimal because according to our definition, it is possible for a field to change its value as long as its containing object is still being constructed and has not been made available for other threads; however, `final` fields can be written to only once in the object constructor.

3.2.1.2 Analysis

The following algorithm computes the set S of immutable fields:

(I) Compute S, the set of all fields of all classes.
(II) Remove from $S$ all fields that are written to in a method which is not a constructor or a static initializer.

(III) Remove from $S$ all instance fields $o.f$ such that an assignment of the form $o.f = x$ exists where $o$ is different than $\text{this}$.

(IV) Remove from $S$ all static fields $C.f$ such that an assignment of the form $C.f = x$ exists in a method which is not a $C$ static initializer method.

(V) Remove from $S$ all fields of classes whose constructors expose $\text{this}$.

To enable the optimization, the set of all immutable fields is computed and stored before the STM instrumentation takes place. When a field access statement is about to be instrumented, we first check if that field is immutable. If so, we avoid instrumenting the field access.

### 3.2.1.3 Applicability

To understand the applicability of the Immutable Memory optimization, we measured for every benchmark, which percentage of all memory reads occurring in a transaction is in fact a read of an immutable memory location. The results are presented in Table 3.1.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% of Reads of Immutable Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>50.0%</td>
</tr>
<tr>
<td>SkipList</td>
<td>65.3%</td>
</tr>
<tr>
<td>Hash</td>
<td>50.4%</td>
</tr>
<tr>
<td>Bank</td>
<td>0.0%</td>
</tr>
<tr>
<td>K-Means</td>
<td>0.0%</td>
</tr>
<tr>
<td>Ssca2</td>
<td>0.0%</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>25.0%</td>
</tr>
<tr>
<td>Vacation</td>
<td>7.6%</td>
</tr>
</tbody>
</table>

Table 3.1: % of reads of immutable memory.

We find that as many as 65.3% of a workload’s memory reads are of immutable data. For example, in LinkedList, many of the accesses are reads of a node’s value, which is set at the node’s construction and never changes. In SkipList and Hash, many reads are of immutable arrays or of constants.

### 3.2.2 Thread Local Memory

#### 3.2.2.1 Introduction

A runtime object $o$ is thread-local if it can be accessed only by the thread that created it. Otherwise, it is thread-shared. Operations on thread-local objects are not in any risk for data race, because such objects may only be accessed from a single thread. However,
operations on thread-local objects still need to be logged: in case of a transaction abort, either programmer-initiated or system-initiated, the object must revert to its pre-transaction state.

Assuming a TL2 STM, there are two valuable optimizations that we can enable when accessing thread-local memory. First, when reading a field off a thread-local object, there is no need to perform expensive pre- and post-validation, since the field’s value cannot have been modified by a different thread. Second, when committing, there is no need to acquire or release any locks to protect thread-local memory. Locking, unlocking, and validating are all operations on shared memory, which require synchronization primitives; by optimizing them away we can gain considerable performance boost.

3.2.2.2 Analysis

In order to detect thread-local objects, we use the characterization of Rountev et al. [71]: an object $o$ is thread-local if it is not reachable (via chains of field references) from any (i) static fields, or (ii) fields of objects of classes that implement the java.lang.Runnable interface. Such objects are thus accessible only by the thread that created them. We conservatively consider all other objects as thread-shared.

In order to track objects throughout the program, we employ a points-to analysis. The points-to analysis is computed as a preliminary step before the thread escape analysis. Our implementation of points-to analysis produces, for each reference-typed variable $v$, a set $pta(v) = \{n_1, n_2, \ldots, n_k\}$ of allocation site nodes. Each node represents a unique new statement in the analyzed program.

Given a node $n \in pta(v)$, which represents an allocation site of an object of type $T$, we can obtain $n.f$ where $f$ is a field of type $T$. $n.f$ is therefore the set of all nodes that $v.f$ may point to.

This is a description of our thread-escape algorithm.

The algorithm generates the set of all nodes that are pointed-to from variables which are either implementors of java.lang.Runnable, or directly accessible from static fields. Then, it computes the closure of that set by the field-reachability relation. The returned set $A$ is therefore a conservative approximation of the set of all thread-shared objects. Its complement is the set of thread-local objects.
### Algorithm 1: Thread Shared Analysis

1: \( A \leftarrow \emptyset \)
2: \( \text{for all method } M \) do
3: \( \text{for all assignment statement } s \text{ in } M \) do
4: \( \text{if } s \text{ is of the form } v.f = o \text{ such that } v \text{ implements } \text{java.langRunnable} \) then
5: \( A \leftarrow A \cup pta(v) \)
6: \( \text{if } s \text{ is of the form } C.f = o \) then
7: \( A \leftarrow A \cup pta(o) \)
8: \( \text{repeat} \)
9: \( \text{for all node } n \text{ in } A \) do
10: \( \text{for all field } f \text{ in } n \) do
11: \( A \leftarrow A \cup n.f \)
12: \( \text{until } A \text{ doesn’t change} \)
13: \( \text{return } A \)

#### 3.2.2.3 Applicability

To understand the applicability of the Thread Local Memory optimization, we measured how many synchronization operations can be removed due to the optimization. Locking and unlocking require a CAS operation, while validating the read-set require reading from the shared locks array. The results are presented in Table 3.2.

<table>
<thead>
<tr>
<th>LinkedList</th>
<th>SkipList</th>
<th>Hash</th>
<th>Bank</th>
<th>K-Means</th>
<th>Ssca2</th>
<th>MatrixMul</th>
<th>Vacation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>3.4%</td>
</tr>
</tbody>
</table>

Table 3.2: % of synchronization operations removed.

We see that accesses to thread-local memory are quite rare. This is an indication that the benchmarks are well-designed as most transactions access only shared memory. Vacation’s thread-local memory accesses are to arrays which govern what random action each thread will perform within the transaction. Other benchmarks don’t access thread-local memory inside transactions.

#### 3.2.3 Transaction Local Memory

##### 3.2.3.1 Introduction

Consider an object \( o \) created inside a transaction. Immediately after its creation, \( o \) is transaction-local: access to it is possible only from within the transaction. In lazy-update STMs, \( o \) remains transaction-local until the transaction is committed; this is
not the case in in-place STMs, where \( o \) may be made visible to other threads even before the transaction is committed.

Since \( o \) is not visible to any other thread, there can be no data race when accessing its fields. And since it is completely disposed of in case of abort, there is no need to log previous or new field values. We can therefore make use of the transaction-local property by skipping the instrumentation of all fields of such objects.

3.2.3.2 Analysis

We provide two complementing analyses that statically discover access operations to fields of transaction-local objects: the first finds accesses to \( \text{this} \) inside constructors, and the second finds accesses to newly-constructed objects inside methods.

**In-Constructor Analysis:** The following algorithm computes the set \( A \) of self accesses to a transaction-local object in its own constructor.

\[
\begin{align*}
\text{Algorithm 2: In-Constructor Transaction-Local Analysis} \\
1: & \quad A \leftarrow \emptyset \\
2: & \quad \text{for all class } C \text{ do} \\
3: & \quad \quad \text{for all constructor } M \text{ of class } C \text{ do} \\
4: & \quad \quad \quad \text{for all statement } s \text{ in } M \text{ of the form } \text{this}.x = v \text{ or } v = \text{this}.x \text{ do} \\
5: & \quad \quad \quad A \leftarrow A \cup s \\
6: & \quad \text{return } A
\end{align*}
\]

Indeed, any read or write access to the \( \text{this} \) reference inside of a constructor is a transaction-local memory access.

**After-Constructor Analysis:** The goal of the algorithm described here is to find the set of accesses to fields of transaction-local objects after their constructor method has finished constructing them. Typically, the fields of the newly-constructed object are initialized shortly after its creation.

We use an intraprocedural, flow-sensitive data flow analysis. The analysis tracks the creation of objects and accesses to fields. It uses the results of a points-to analysis that was pre-calculated. A points-to analysis associates every memory access expression \( e \) (of the form \( o.f \) or \( a[i] \)) with a set of abstract memory locations, \( pta(e) \subseteq L \). Points-to analysis guarantees that two expressions \( e_1 \) and \( e_2 \) do not access the same memory location if and only if \( pta(e_1) \cap pta(e_2) = \emptyset \).

The set of tags \( T \) is composed of the following elements: \( \bot \) (memory location is not transaction-local), \( NEW \) (memory location is transaction-local), and \( \top \) (memory location...
Each statement $s$ in the program induces two program points, $\cdot s$ and $s \cdot$. The analysis associates each program point $p$ with a set $S(p) \subseteq L \times T$ of tagged abstract memory locations. For a statement $s$, $S(\cdot s)$ represents the knowledge of the analysis just before executing $s$, and $S(s \cdot)$ represents the knowledge of the analysis just after executing $s$.

Each program point is initially associated with the full set of abstract memory locations, and each memory location is tagged $\top$.

The algorithm generates information in the following way. For each statement $s$, if it does not read from or write to any memory location, we set $S(\cdot s) \leftarrow S(\cdot s)$. Otherwise, let $e$ be the memory access expression. We compute the set of pointed-to abstract memory locations $pta(e)$. For every $l \in pta(e)$ and $< l, t > \in S(\cdot s)$, we compute and set
\[
S(s \cdot) \leftarrow (S(s \cdot) \setminus \{< l, t >\}) \cup \{< l, \tau(s) >\} \tag{3.1}
\]

The transfer function $\tau$ inspects $s$. If $s$ is an object creation expression, it returns $NEW$. If $s$ is a copy statement, or a primitive field store or load (e.g., $s$ is of the form $o.f = i$ or $i = o.f$ where $i$ is a primitive), $\tau$ returns $t$. For all other statement types $s$, $\tau$ conservatively returns $\bot$.

Let $pred(s)$ be the set of statement $s$’s predecessors. The information is propagated forward according to the data flow equations defined by:
\[
S(\cdot s) \leftarrow \{< l, \arg \min_{t'} \{< l, t' > \in S(p \cdot) : p \in pred(s)\} : l \in L\} \tag{3.2}
\]

The static analysis algorithm iterates over the program’s Control Flow Graph (CFG) and applies equations 3.1, 3.2 until a fixed point is reached. This concludes the analysis.

Finally, we use the results of the analysis as follows. Consider a program statement $s$ which accesses memory access expression $e$. Let $t = \min \{t' : < l, t' > \in S(\cdot s), l \in pta(e)\}$ be the minimum tag found among all memory locations potentially pointed to by $e$. If $t = NEW$ then we can access $e$ without any instrumentation, since the object pointed-to by it was created in the same transaction.

### 3.2.3.3 Applicability

We measured, for every benchmark, what percentage out of the total number of memory accesses are discovered as transaction-local accesses. We show the total of the two
complementing analyses together. The results are presented in Table 3.3.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% of transaction-local accesses detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>0.03%</td>
</tr>
<tr>
<td>SkipList</td>
<td>0.4%</td>
</tr>
<tr>
<td>Hash</td>
<td>0.0%</td>
</tr>
<tr>
<td>Bank</td>
<td>0.0%</td>
</tr>
<tr>
<td>K-Means</td>
<td>0.0%</td>
</tr>
<tr>
<td>Ssca2</td>
<td>0.0%</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>0.0%</td>
</tr>
<tr>
<td>Vacation</td>
<td>3.2%</td>
</tr>
</tbody>
</table>

Table 3.3: % of transaction-local accesses detected.

Many benchmarks, such as MatrixMul and K-Means, do not allocate any memory inside transactions; this optimization is irrelevant for them, and they gain nothing from it. Other benchmarks allocate memory inside transactions. Vacation’s transactions, for example, add nodes to a red-black tree. According to our results, 3.2% of their memory accesses can be performed without any instrumentation.

It is especially revealing to measure the amount of memory writes that are transaction-local: the numbers are as high as 66.6% for LinkedList, 8.9% for SkipList, and 49.3% for Vacation. The high numbers demonstrate that this optimization is indeed strongly applicable to our benchmarks and suggest that it can be effective. We note that the numbers in Table 3.3 are much lower; this is because the table measures accesses (reads and writes) rather than just writes, and these benchmarks perform significantly more reads than writes. It is much more common to write to a transaction-local memory location than it is to read from it.

3.3 STM Optimizations — New Techniques

3.3.1 Read-Only Transactions

3.3.1.1 Introduction

A read-only transaction is a transaction which does not modify shared memory at all. Read-only transactions can execute with less overhead than a general transaction, because they do not need to populate and validate a read-set [22]. This can represent a significant saving. Consider, for example, a typical linked list’s contains() transaction, which traverses the list while looking for an item. If we detect that contains() is read-only, we can skip needless insertion of many of the list’s elements to the read-set.

We provide a static analysis that detects read-only transactions.
3.3. STM OPTIMIZATIONS — NEW TECHNIQUES

3.3.1.2 Analysis

Let method \( f \) be in scope if \( f \) is marked as an atomic method, or it is reachable (via some chain of method invocations) from an atomic method. In the following analysis, we consider only methods in scope.

Our goal is to detect and mark methods which are statically read-only, so they can be exempt from read-set population. By static we mean that it is safe to skip read-set population in them for every call context in which they are invoked. In other words, a read-only method participates only in read-only transactions; otherwise it would be wrong to skip read-set population in it, since it might be called from a non-read-only transaction.

Our algorithm is based around propagating the “non-read-only” property of methods in scope across the program’s call graph. We begin by assigning the non-read-only property to all locally non-read-only methods: those methods which directly write to memory. The propagation is done according to the two following rules:

1. If a method \( f \) is not read-only, then so are all methods \( g \) that it invokes.
2. If a method \( f \) is not read-only, then so are all methods \( g \) which invoke it.

The reason for rule 1 is that any transaction which invokes \( f \) might invoke \( g \), thereby making \( g \) participate in a non-read-only transaction. Similarly, the reason for rule 2 is that any transaction which invokes \( g \) might invoke \( f \), thereby making \( g \) participate in a non-read-only transaction.

We use a simple worklist to do the propagation, starting from locally non-read-only methods. The analysis ends when the propagation hits a fixed-point. At that point, all the methods which are not marked as non-read-only are considered read-only, and can be marked as methods for which read-set population can be avoided.

3.3.1.3 Applicability

To understand the applicability of the Read Only Transaction optimization, we optimized each of the benchmarks, and measured how many memory reads are reads that do not have to log their value to the read-set. The percentages are displayed in Table 3.4.

Our analysis was able to detect LinkedList’s \texttt{contains()} method, SkipList’s \texttt{contains()} method, and Bank’s \texttt{computeTotal()} method as read-only methods. The three methods do not alter shared memory and can therefore skip inserting elements
Table 3.4: % of reads from memory not changed within the transaction.

<table>
<thead>
<tr>
<th></th>
<th>LinkedList</th>
<th>SkipList</th>
<th>Hash</th>
<th>Bank</th>
<th>K-Means</th>
<th>Ssca2</th>
<th>MatrixMul</th>
<th>Vacation</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.6%</td>
<td>85.9%</td>
<td>0.0%</td>
<td>82.6%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

to the read-set. We could not detect Hash’s `contains()` method as read-only, even though it is in fact read-only, because it invokes an auxiliary method `index()` that searches the hash table. `index()` is used by other Hash methods, and could not be inlined in `contains()`.

### 3.3.2 Load Elimination

#### 3.3.2.1 Introduction

Consider the following code (Figure 3.2) fragment that is part of an atomic block (derived from the Java™ version of the STAMP suite):

```java
1   for (int j = 0; j < nfeatures; j++) {
2       newCenters[index][j] = newCenters[index][j] + feature[i][j];
3   }
```

Figure 3.2: Redundant load.

A naïve STM compiler will instrument every array access in this fragment. However, the memory locations `newCenters[index]` and `feature[i]` are loop-invariant. We can calculate them once, outside the loop, and re-use the calculated values inside the loop.

The technique of re-using values is a form of Partial Redundancy Elimination (PRE) optimization and is common in modern compilers. When PRE is applied to memory loads, it is called *Load Elimination*. The optimized version of the code will be equivalent to code shown in Figure 3.3.

```java
1   if (0 < nfeatures) {
2       nci = newCenters[index];
3       fi = feature[i];
4       for (j = 0; j < nfeatures; j++) {
5           nci[j] = nci[j] + fi[j];
6       }
7   }
```

Figure 3.3: Eliminate redundant load.
Many compilers refrain from applying the technique to memory loads (as opposed to arithmetic expressions). One of the reasons is that such a code transformation may not be valid in the presence of concurrency; for example, the compiler must make sure that the feature memory location cannot be concurrently modified by a thread other than the one executing the above loop. This constraint, however, does not exist inside an atomic block, because the atomic block guarantees isolation from other concurrent transactions. An STM compiler can therefore enable PRE optimizations where they would not be possible with a regular compiler that does not support atomic blocks.

We note that this optimization is sound for all STM protocols that guarantee isolation. The performance boost achieved by it, however, is maximized with lazy-update STMs as opposed to in-place-update STMs. The reason is that lazy-update STMs must perform an expensive write-set lookup for every memory read, while in in-place-update STMs, memory reads are relatively cheap since they are done directly from memory. In fact, in such STMs, reads of memory locations that were read before can be optimized (Cheng et al. [24, 86]) to perform just a direct memory read together with a consistency check. By using load elimination of memory locations, memory reads are transformed into reads of a local variable; as a result, such reads are made much faster and a lazy-update STM operates at in-place STM speeds.

### 3.3.2.2 Analysis

We follow the classic Lazy Code Motion (LCM) ([49]) algorithm. This algorithm computes a few data flow analyses, and uses them to remove redundancies by assigning computed values into temporary variables and reusing them later. We apply LCM to field and array references, as well as to scalar expressions. For field and array references, we use a side-effect analysis to discover if the values do not change from one read to another. For example, consider the fragment \( v1 = o1.f; o2.f = \ldots; v2 = o1.f; \). Can we eliminate the load into \( v2 \) by reusing the value of \( o1.f \) that was previously loaded into \( v1 \)? The answer depends on whether the store to \( o2.f \) may modify \( o1.f \) (that is, whether \( o1 \) and \( o2 \) may point to the same memory location); If it can, then it is not safe to eliminate the load to \( v2 \). The side-effect analysis provides us with answers to such questions.
3.3.2.3 Applicability

Table 3.5 shows the reduction in the number of instrumented memory reads as a result of applying our algorithm.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>0.0%</td>
</tr>
<tr>
<td>SkipList</td>
<td>27.6%</td>
</tr>
<tr>
<td>Hash</td>
<td>7.8%</td>
</tr>
<tr>
<td>Bank</td>
<td>1.2%</td>
</tr>
<tr>
<td>K-Means</td>
<td>58.9%</td>
</tr>
<tr>
<td>Ssca2</td>
<td>0.0%</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>25.1%</td>
</tr>
<tr>
<td>Vacation</td>
<td>1.8%</td>
</tr>
</tbody>
</table>

Table 3.5: % of reads eliminated after applying PRE.

We see that some benchmarks (SkipList, K-Means, MatrixMul) contain many redundancies which can be removed. The PRE technique is ineffective on small and loop-less transactions (Ssca2) and on tightly written code with no redundancies (LinkedList). The advantage of PRE is that it does not require any modifications to the STM library functions.

3.3.3 Scalar Promotion

3.3.3.1 Introduction

The dual to load elimination is Scalar Promotion. Consider the following code fragment (Figure 3.4), also from STAMP.

```c
1   for (int i = 0; i < num_elts; i++) {
2       moments[0] += data[i];
3   }
```

Figure 3.4: Redundant writes.

If this fragment appeared inside an atomic method, an STM compiler could take advantage of the isolation property to eliminate the multiple writes to the same memory location. An optimized version of the code would be equivalent to the code shown in Figure 3.5.

The advantage of the optimized version is that multiple memory writes are replaced with just one.

3.3.3.2 Analysis

Our scalar promotion algorithm searches for loops in which a field or array reference is written to. For each written reference, it attempts to discover whether it is promotable.
3.3. STM OPTIMIZATIONS — NEW TECHNIQUES

1 if (0 < num_elts) {
2    double temp = moments[0];
3    try {
4        for (int i = 0; i < num_elts; i++) {
5            temp += data[i];
6        }
7    } finally {
8        moments[0] = temp;
9    }
10 }

Figure 3.5: Eliminate redundant writes.

We will present the algorithm for field references; promoting array references is done in a similar manner. Consider, for example, a statement o.f = ... contained inside a loop. o.f is promotable if the two following conditions are satisfied: 1. o is never modified inside the loop, and 2. All writes to o.f are assignment statements of the exact form o.f = .... The second condition is needed to invalidate the promotion in case o.f is modified by a method call or by a reference that points to the same object as o. In case o.f is found promotable, we transform the loop as follows: we add a preheader, in which we read o.f’s value into a new variable, t; we add a tail in which we store t’s value to o.f; and we replace all occurrences of o.f in the loop by t.

3.3.3.3 Applicability

Our analysis could not locate any instances where scalar promotion can be used. By examining the benchmarks code, we could not find examples where it was obvious that the same memory location was written to more than once. We expect that real production code will contain at least a few instances where this optimization is applicable; however this will occur at a much lower frequency than instances where PRE and other optimizations are possible.

3.3.4 Redundant Write-Set Lookups

3.3.4.1 Introduction

Consider a field read statement v = o.f inside a transaction. The STM must produce and return the most updated value of o.f. In STMs that implement lazy update, there can be two ways to look up o.f’s value: if the same transaction has already
written to \texttt{o.f}, then the most updated value must be found in the transaction’s write-set. Otherwise, the most updated value is the one in \texttt{o.f}'s memory location. A naïve instrumentation will conservatively always check for containment in the write-set on every field read statement. With static analysis, we can gather information whether the accessed \texttt{o.f} was possibly already written to in the current transaction. If we can statically deduce that this is not the case, then the STM may skip checking the write-set, thereby saving processing time.

### 3.3.4.2 Analysis

We use an interprocedural, flow-sensitive data flow analysis. The analysis tracks the memory locations that are accessed by transactions, and their usage patterns. It can be seen as a compile-time simulation of the read-set and write-set run-time activity.

The analysis uses the results of a \emph{points-to analysis} that was pre-calculated. A points-to analysis associates every memory access expression $e$ (of the form \texttt{o.f} or \texttt{a[i]}) with a set of abstract memory locations, $pta(e) \subseteq L$. Points-to analysis guarantees that two expressions $e_1$ and $e_2$ do not access the same memory location if and only if $pta(e_1) \cap pta(e_2) = \emptyset$.

The set of tags $T$ is composed of the following elements: ⊥ (memory location was written to), $RO$ (memory location was read at least once, but not written to thereafter), and $\top$ (memory location was not accessed yet). The tags are ordered: $\bot \leq RO \leq \top$.

Each statement $s$ in the program induces two \emph{program points}, $\cdot s$ and $s \cdot$. The analysis associates each program point $p$ with a set $S(p) \subseteq L \times T$ of tagged abstract memory locations. For a statement $s$, $S(\cdot s)$ represents the knowledge of the analysis just before executing $s$, and $S(s \cdot)$ represents the knowledge of the analysis just after executing $s$.

Each program point is initially associated with the full set of abstract memory locations, and each memory location is tagged $\top$.

The algorithm generates information in the following way. For each statement $s$, if it does not read from or write to any memory location, we set $S(\cdot s) \leftarrow S(\cdot s)$. Otherwise, let $e$ be the memory access expression. Let $a \leftarrow \textit{Read}$ if $s$ reads from $e$, otherwise $a \leftarrow \textit{Write}$. We compute the set of pointed-to abstract memory locations $pta(e)$. For every $l \in pta(e)$ and $< l, t > \in S(\cdot s)$, we compute and set

$$S(s \cdot) \leftarrow (S(s \cdot) \setminus \{< l, t >\}) \cup \{< l, \tau(t, a) >\}$$

(3.3)
3.3. STM OPTIMIZATIONS — NEW TECHNIQUES

where the transfer function $\tau$ is defined in the Figure 3.6. $\tau(t, a)$ is the node reachable by stepping from node $t$ across the edge $a$.

![Diagram of the transfer function $\tau$.](image)

Figure 3.6: Transfer function $\tau$.

Let $\text{pred}(s)$ be the set of statement $s$’s predecessors. The information is propagated forward according to the data flow equations defined by:

$$S(·s) ← \{< l, \arg \min_{t' \in \text{pred}(s)} \{< l, t' > \in S(p) : p \in \text{pred}(s) \} : l \in L \}. \quad (3.4)$$

There is one exception to this propagation equation. if $·s$ is a program point just before starting a new transaction, and all its predecessors are not part of a transaction, then we propagate $\top$ tags instead of the minimum tag seen over all predecessors. The reason is that when starting a transaction, the read-set and write-set and empty.

The static analysis algorithm iterates over the program’s Control Flow Graph (CFG) and applies equations 3.3, 3.4 until a fixed point is reached. This concludes the analysis.

Finally, we use the results of the analysis as follows. Consider a program statement $s$ which reads memory access expression $e$. Let $t = \min \{t' : < l, t' > \in S(·s), l \in \text{pta}(e) \}$ be the minimum tag found among all memory locations potentially pointed by $e$. If $t = RO$ then we know that in the transaction surrounding $s$, no memory location pointed to by $e$ has been written to before the execution of $s$. Therefore when executing $s$ there is no need to lookup $e$’s value in the write-set.

3.3.4.3 Applicability

We measured the percentage, for each benchmark, of reads from memory locations, such that these memory locations have not been written to before in the same transaction. The results are presented in Table 3.6.

Our analysis was successful on almost all benchmarks. Indeed, transactions usually
read memory before writing to it. As an example of this optimization, consider how `add()` and `remove()` work in LinkedList, SkipList and Hash: They first locate the position to insert or remove the element, do the insertion or removal and finally perform some structural fixes. The first step (locating the position) encompasses the majority of memory reads, and since it precedes any memory writes, the optimization is applicable to it.

### 3.3.5 Redundant Write-Set Record-Keeping

#### 3.3.5.1 Introduction

Consider a field write statement `o.f = v` inside a transaction. An STM with lazy-update protocol must update the write-set with the information that `o.f` has been written to. One of the design goals of the write-set is that it should be fast to search it; this is because subsequent reads from `o.f` in the same transaction must use the value that is in the write-set. But, some memory locations in the write-set will never be actually read in the same transaction. We can exploit this fact to reduce the amount of record-keeping that the write-set data-structure must handle. As an example, TL2 suggests implementing the write-set as a linked-list (which can be efficiently added-to and traversed) together with a Bloom filter (that can efficiently check whether a memory location exists in the write-set). If we can statically deduce that a memory location is written-to but will not be subsequently read in the same transaction, we can skip updating the Bloom filter for that memory location. This saves processing time, and is sound because there is no other purpose in updating the Bloom filter except to help in rapid lookups.

#### 3.3.5.2 Analysis

Consider a statement `s` that writes to a memory access expression `e`. In order to find out whether there is no subsequent statement which reads from any of the memory locations pointed to by `e`, we create an “opposite” version of the previous analysis.

<table>
<thead>
<tr>
<th>LinkedList</th>
<th>SkipList</th>
<th>Hash</th>
<th>Bank</th>
<th>K-Means</th>
<th>Ssca2</th>
<th>MatrixMul</th>
<th>Vacation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>98.0%</td>
<td>100.0%</td>
<td>88.4%</td>
<td>80.2%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>3.9%</td>
</tr>
</tbody>
</table>

Table 3.6: % of reads from locations not written to before.
The opposite version propagates information backwards rather than forwards. Its set of tags consists of: \( \bot \) (memory location was read), \( WO \) (memory location was written to at least once, but not read thereafter), and \( \top \) (memory location was not accessed yet). The tags are ordered: \( \bot \leq WO \leq \top \). We omit the rest of the details since they are very similar to the previous analysis.

### 3.3.5.3 Applicability

We measured the percentage, for each benchmark, of writes to memory locations, such that these memory locations will not be subsequently read from in the same transaction. The results are presented in Table 3.7.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>100.0%</td>
</tr>
<tr>
<td>SkipList</td>
<td>4.2%</td>
</tr>
<tr>
<td>Hash</td>
<td>100.0%</td>
</tr>
<tr>
<td>Bank</td>
<td>10.0%</td>
</tr>
<tr>
<td>K-Means</td>
<td>5.3%</td>
</tr>
<tr>
<td>Ssca2</td>
<td>100.0%</td>
</tr>
<tr>
<td>MatrixMul</td>
<td>100.0%</td>
</tr>
<tr>
<td>Vacation</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

Table 3.7: % of writes to locations which will not be subsequently read from.

We see that a majority of writes could in fact be detected as writes to memory locations which will not be read from in the same transaction. As a counter-example, consider Bank’s `addInterest()` method, which atomically updates all the accounts. Each memory write, that updates the balance, is indeed not followed by a read of the same balance; however, the points-to analysis cannot distinguish between the multiple accounts. The result is that the analysis thinks that the same balance is read, written, re-read and re-written again and again; so the optimization is not applicable to the balance update.

### 3.3.6 Tightening Transaction Scopes

#### 3.3.6.1 Introduction

The goal of this optimization is to shorten the lifetime of transactions. Consider the `minPair()` method at listing 3.7. This method atomically reads the two items of the incoming pair, and then computes their minimum. Assume that this method is never called from an atomic context. The statements where the minimum is computed need not, in fact, belong in a transaction: they contain no field accesses, array accesses, or method invocations. A transaction boundaries analysis will deduce that the instructions following `int y = p.y;` can be hoisted outside of the transaction.
CHAPTER 3. LOWERING STM OVERHEAD WITH STATIC ANALYSIS

Figure 3.7: Tightening transaction scope.

Using transaction boundaries information, we can shorten the lifetime period when a transaction is active, by committing earlier. In the example, we can commit immediately after `int y = p.y;`, instead of waiting for the method to return to its caller and then commit (recall that in the naïve instrumentation, every atomic method is wrapped by a retry-loop, where the actual commit is performed). Shortening transactions helps in reducing contention: conflicts can be detected earlier, and less amount of wasted work would be done if the transaction is doomed to abort.

Symmetrically, we can find opportunities where we can shorten the transaction by initializing it later. This can happen, for example, if a transaction’s first action is to allocate memory (e.g. allocate a new node for a linked-list). This allocation can happen outside the scope of the transaction. In TL2, a transaction begins by reading the shared global clock. Initializing the transaction later means that we may read a later value of the clock, thereby possibly avoiding collisions with other committing transactions.

3.3.6.2 Analysis

Let $m$ be an atomic method. We seek to find all initialization points (IPs): program points in $m$ where it is safe to start the transaction. (Finding commit points requires minor modifications that “reverse” to algorithm; we omit the details.) We require that: 1. all (runtime) paths through $m$ go through exactly one IP, 2. all (runtime) paths through $m$ go through an IP before going through any other STM library calls, and 3. the IPs are placed as late as possible. Once we have calculated the set of initialization points, we can optimize the instrumentation of $m$ as follows: 1. remove the STM library’s initialization call from $m$’s retry-loop, and 2. insert STM library initialization calls in all IPs.

To make the “as late as possible” point concrete, we list the program statements that require no STM instrumentation and therefore can appear on a path before an IP. These are: 1. statements that do not access memory directly or indirectly; 2. statements that access immutable memory locations; 3. statements that access memory locations which
were allocated in the same transaction as \( m \); 4. statements which invoke methods which require no STM instrumentation.

The algorithm that finds IPs is composed of three major steps and operates on the strongly-connected components (SCC) graph of \( m \). We use SCCs because IPs may never be placed inside loops.

In the first step, we detect all statements that must be instrumented in the program. Then, for every SCC of size larger than 1 that contains a must-instrument statement, we mark all its SCC predecessors as requiring instrumentation. This process repeats until we reach a predecessor SCC of size 1 (this is possibly \( m \)’s head).

The second step is a simple forward intraprocedural data flow analysis over \( m \)’s SCC graph. For every SCC \( s \), it associates its two program points \((\cdot s \text{ and } s\cdot)\) with a tag \( t \in T \). \( T \)’s elements are: \( \bot \) (possibly initialized already), \( DI \) (definitely initialized already), and \( \top \) (not initialized yet). Every SCC program point starts associated with \( \top \). Every must-instrument SCC generates \( DI \). Flow merge points generate \( \bot \) if the tags corresponding to the incoming branches are not all the same, otherwise their common value. We propagate until convergence.

In the final step, let \( S(p) \) be the tag associated with a program point \( p \). Consider all SCCs \( s \) such that \( S(s\cdot) = DI \). If \( S(\cdot s) = \top \) then we mark \( s \) as an initial initialization point (IIP). If \( S(\cdot s) = \bot \) then we mark \( s \) as a recurring initialization point (RIP).

After IIPs and RIPs have been calculated, the optimization proceeds by injecting calls to the STM library transaction initialization method at the beginning of every SCC marked IIP. The first step guarantees that such SCCs contain exactly one statement, so the injection placement is unambiguous. The second step guarantees that IIP program points are reached only by (runtime) paths that did not already initialize the transaction.

RIPs represent program points such that some paths leading to them may have initialized the transaction and some others may have not; therefore at every RIP we inject a conditional initialization statement, that initializes the transaction only if it was not already initialized. This is accomplished by using a local boolean flag. In the presence of RIPs we must set this flag at all IIPs; if the analysis did not turn up any RIPs, the flag is not needed. As before, RIPs are only possible at SCCs of size 1.
3.3.6.3 Applicability

To measure the applicability of the transaction scope tightening optimization, we examined only cases where memory access operations could be hoisted outside of transactions—we ignored cases where only other instructions (i.e., arithmetic, branching or memory allocations) could be hoisted. This allows us to meaningfully measure and compare this optimization to other optimizations. Also, for this optimization to be effective, it must use information about immutable and transaction-local; so these two analyses were performed as well. We measure which percentage of memory reads could be hoisted outside transactions. The results appear in Table 3.8.

<table>
<thead>
<tr>
<th>LinkedList</th>
<th>SkipList</th>
<th>Hash</th>
<th>Bank</th>
<th>K-Means</th>
<th>Ssca2</th>
<th>MatrixMul</th>
<th>Vacation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.03%</td>
<td>7.1%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.03%</td>
</tr>
</tbody>
</table>

Table 3.8: % of reads hoisted outside of transactions.

The most visible gain was found in SkipList, where we discovered 3 initialization points and 1 commit point. Other benchmarks gained next to nothing. The low applicability of this optimization is due to two reasons: (1) Transactions with lengthy loops gain very little from optimizing just few instructions outside the loops. (2) The transactions that appear in the benchmarks are already tight. We note that our benchmarks did not contain any instances of RIP or recurring commit points.

3.4 Performance Evaluation

In this section we present a performance evaluation of the different optimizations. Our test environment is a Sun UltraSPARC™ T2 Plus multicore machine.

3.4.1 Optimization Levels

We compared nine levels of optimizations. The levels are cumulative in that every level includes all the optimizations of the previous levels.

(I) **None** level is the most basic code, which blindly instruments every memory access.

(II) **+Immutable** level avoids instrumentation of accesses to immutable memory (Section 3.2.1).
(III) +TransactionLocal level avoids instrumentation of accesses to transaction-local memory (Section 3.2.3).

(IV) +ThreadLocal level avoids synchronization instructions when accessing thread-local memory (Section 3.2.2).

(V) +ReadOnlyMethod level avoids read-set population in read-only transactions (Section 3.3.1).

(VI) +PRE level performs a load elimination optimization (Section 3.3.2).

(VII) +write-setLookup level avoids redundant write-set lookups for memory locations which have not been written to (Section 3.3.4).

(VIII) +write-setStorage level avoids redundant write-set record-keeping for memory locations which will not be read (Section 3.3.5). Finally,

(IX) +ScopeTightening level shortens transactions by hoisting out instructions (Section 3.3.6).

3.4.2 Benchmarks

We experimented on a set of microbenchmarks and several benchmarks from the Java™ version [21] of the STAMP [16] suite. The results for applicability were obtained on a single-threaded run of the benchmarks.

3.4.2.1 Microbenchmarks

The data microbenchmarks benchmarks consist of three different data structures: LinkedList represents a sorted linked list implementation, SkipList represents a skiplist with random leveling, Hash represents an open-addressing hash table which uses a fixed-size array with no rehashing. Every data structure supports three atomic operations: adding an item, removing an item, and checking for containment of an item. The test consists of threads attempting to perform as many atomic operations as possible on a shared data structure; each thread chooses its next operation randomly, with 90% chance selecting the lookup operations, and 10% chance selecting addition or removal.
In addition, the microbenchmarks include **Bank**, an online bank simulation, in which different threads perform either a read-all transaction with 80% chance, a write-all transaction with 10% chance, or a transfer between two accounts with 10% chance.

The graphs in Figure 3.8 depict normalized throughput of these benchmarks. The graphs show normalized throughput as a function of both the number of threads (the x-axis) and the optimization level applied (the different columns for the same x mark). Since we measure throughput, the higher the bar is, the better.

**Figure 3.8: Microbenchmarks comparative results. (higher is better)**

**LinkedList** benefited mostly from 3 optimizations. First, skipping instrumentation of immutable memory increased throughput dramatically: up to 82% in the single-threaded run and up to 73% in the 64-thread run. The ReadOnlyMethod optimization, which skips population of the read-set in the contains() method, increased throughput by up to 14% (16 threads). Finally, the write-setLookup optimization, which is applicable to all of LinkedList’s transactioned memory reads, increased throughput by 46-53% (4-16 threads).

**SkipList** also benefited mostly from these 3 optimizations. Immutable boosted throughput by 95-125% (1-64 threads). ReadOnlyMethod increased throughput
by 5-13% (32-64 threads). The write-setLookup optimization increased throughput by around 20%. The effect of the ScopeTightening optimization was noticeable only in the 64-thread run, where it increased throughput by 8%. PRE increased throughput by 1-3%.

Hash saw almost no benefit from the optimizations at 1-4 threads. At higher amounts of threads, the most effective optimization was Immutable - up to 12-17% improvement.

Bank did not expose many opportunities for optimizations. The highest gain was from write-setLookup (6-19% improvement at 1-16 threads). Even though the computeTotal() function was discovered as a ReadOnlyMethod, there was not much improvement (up to 4% at 8 threads) because the read-set size was very low (equal to the number of threads). Bank is the least scalable benchmark because even one transaction performing addInterest() will conflict and stall all other transactions.

3.4.2.2 STAMP Benchmarks

The STAMP benchmarks consist of: K-Means implements k-means clustering, Vacation simulates an on-line travel reservation system, Ssca2 performs several graph operations and MatrixMul performs matrix multiplication. We could not test the other benchmarks from STAMP due to technical limitations (they were written in a special dialect of Java [21] and after conversion to standard Java™ they run with incorrect results due to bugs in the benchmarks implementations).

The graphs in Figure 3.9 depict normalized running time of these benchmarks. Similar to the data structure benchmarks, the graphs show normalized running time as a function of both the number of threads and the optimization level applied. In this case we measure running time, so the lower the bar is, the better.

K-Means benefited greatly (28-38% speedup on all thread amounts) from PRE; the example from section 3.3.2 was taken directly from K-Means. Other optimizations were not successful in improving the runtime.

Vacation benefited from 2 main optimizations. The Immutable optimization sped up the benchmarks by 8-19% at all thread amounts. The TransactionLocal optimization, which avoids instrumentation of accesses to transaction-local memory,
sped up the benchmarks by 3-9%. This makes sense, since this optimization was applicable to only 3.2% of the memory accesses. The ThreadLocal optimization, which removed 3.4% of the synchronization operations, was effective only at high thread counts, where it sped up the tests by 3-9% (16-64 threads).

**Ssca2** benefited mostly from two optimizations. The write-setLookup optimization, which as we saw was applicable to all the read accesses, shortened the benchmark times by 2-5% (1-8 threads). The write-setStorage optimization, applicable to all the write accesses, shortened the times by 0-8% (1-32) threads. Even though all the reads and writes were optimized, Ssca2 showed only modest improvement; this is because in Ssca2, transactions are short-lived with relatively small read-sets and write-sets [16].

**MatrixMul**, the most scalable benchmark, was amenable to 4 optimizations: Immutable (30-54% faster), PRE (3-43% faster), write-setLookup (12-22% faster), and write-setStorage (up to 4% faster).

Figure 3.9: STAMP Benchmarks comparative results. (lower is better)
3.4.3 Discussion

By examining the results we see that, among the well-known optimizations, the most impressive gains came from Immutable. The TransactionLocal and ThreadLocal optimizations have low applicability and therefore rarely had a noticeable effect. These results are consistent with previous research \[1\].

Among the new optimizations, the three most powerful optimizations were write-setLookup (up to 53% more throughput), PRE (up to 38% speedup), andReadOnlyMethod (up to 14% more throughput).

The impact of the write-setStorage optimization was relatively small. The reason that this optimization is less effective is that the insertion to the Bloom filter is already a very quick operation. In addition, our tested workloads have a relatively low amount of writes. The impact of ScopeTightening was also small, due to less applicability – transactions in the benchmarks were already tight.

The effectiveness of the optimizations varied widely with the different workloads and benchmarks. For example, the fully optimized LinkedList was 223% faster than the unoptimized version on 1 thread, and 106% faster on 16 threads. MatrixMul was 68% faster on a single-threaded run. However, Ssca2 showed only modest improvements on any number of threads due to short and small transactions.

Finally, we note that at some workloads, the optimizations produced a negative impact on performance. For example, the optimized version of LinkedList at 64 threads performed worse than the non-optimized version. We suspect that, on some specific workloads, making some transactions faster could generate conflicts with other advancing transactions. Despite such isolated cases, the results are encouraging: new optimizations like write-setLookup, PRE and ReadOnlyMethod were successful at significantly saving time and increasing throughput on most workloads.

3.5 Conclusion

We described, implemented and tested a selection of STM-specific optimizations. The optimizations are suitable for every STM protocol, but their effectiveness is greater on lazy-update STMs.

In addition to implementing well-known optimizations, we presented some new ones. First, we provided an analysis to discover read-only transactions. Second, we showed
that two pre-existing optimizations, load elimination and scalar promotion, can be used in an optimizing STM compiler. Where standard compilers need perform an expensive cross-thread analysis to enable these optimizations, an STM compiler can rely on the atomic block’s isolation property to enable them. Third, we highlighted two redundancies in STM read and write operations, and showed how they can be optimized. Last, we showed how to shorten transactions by hoisting statements outside of atomic blocks where it is safe to do so.

We implemented a compiler pass that performs these STM-specific code motion optimizations, and another pass that uses static analysis methods to discover optimization opportunities for redundant STM read and write operations. We have augmented the interface of the underlying STM compiler, Deuce, to accept information about which optimizations to enable at every STM library method call, and modified the STM methods themselves to apply the optimizations when possible.

The combined performance benefit of all the optimizations presented here varies with the workload and the number of threads. While some benchmarks see little to no improvement (e.g., SSCA2), we have observed speedups of up to 38% and throughput increases of up to 46-53% in other benchmarks.

There are many ways to improve upon this research. For example, a drawback of the optimizations presented here is that they require full interprocedural analysis to make sound decisions. It may be interesting to research which similar optimizations can be enabled with less analysis work, for example, with running only intraprocedural analyses, or with partial analysis data that is calculated at runtime.
Chapter 4

Scalable Producer-Consumer Pools
4.1 Overview

Before explaining how the ED-Tree works, let us review its predecessors, the diffracting tree, and elimination tree and the use of elimination and diffraction techniques in building scalable data structures.

4.1.1 Diffracting Trees

Diffracting trees [76] were designed to provide shared counting and load balancing in a distributed parallel environment, and originated from Counting networks [11].

Consider a binary tree of objects called balancers with a single input wire and two output wires, as depicted in Figure 4.1. Threads arrive at a balancer and it repeatedly sends them up and down, so its top wire always has the same or at most one more than the bottom one. The Tree[k] network of width k is a binary tree of balancers constructed inductively by taking two Tree[k/2] networks of balancers and perfectly shuffling their outputs [76].

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{balancer.png}
\caption{A Balancer.}
\end{figure}

A tree constructed in that manner maintains the step property: In any quiescent state, meaning, when every thread that entered the tree arrived at it’s output wire, and there are no threads in inner levels of the tree, the following is correct: if we call the output wires of the tree \(y_1 \ldots y_n\) then either \(y_i = y_j\) for all \(i, j\), or there exists some \(c\) such that for any \(i < c\) and \(j \leq c\), \(y_i - y_j = 1\). [76] In other words, the upper output wires will always have the same number of exited threads as the bottom ones, or at most one more. Thus, the diffracting three spreads the threads in a uniform manner across the output wires. See Figure 4.2.

One could implement the balancers in a straightforward way using a bit that threads toggle: they fetch the bit and then complement it (a CAS operation), exiting on the output wire they fetched (zero or one).

The bad news is that the above implementation of the balancers using a bit means that every thread that enters the tree accesses the same bit at the root balancer, causing that balancer to become a bottleneck. This is true, though to a lesser extent,
4.1. OVERVIEW

with balancers lower in the tree. We can parallelize the tree by exploiting a simple observation:

If an even number of threads pass through a balancer, the outputs are evenly balanced on the top and bottom wires, but the balancer’s state remains unchanged.

Thus, if one could have pairs of tokens collide and then diffract in a coordinated manner, one to the top wire and one to the bottom wire both could leave the balancer without even having to toggle the shared bit. This bit will only be accessed by processors that did not collide. This approach can be implemented by adding a software prism in front of the toggle bit of every balancer. The prism is an inherently distributed data structure that allows many diffractions to occur in parallel. When a thread T enters the balancer, it first selects a location, l, in prism uniformly at random. T tries to “collide” with the previous token to select l or, by waiting for a fixed time, with the next token to do so. If a collision occurs, both tokens leave the balancer on separate wires; otherwise the undiffracted token T toggles the bit and leaves accordingly. Diffracting trees thus provide high degree of parallelism, allowing many threads reach their destination without passing spots of contention.

Although diffracting trees in many cases solve the contention problem, they have some parameters that are sensitive to the number of threads that access the data structure in parallel [75]. One such parameter is the depth of the tree (number of tree levels). If the tree is too small, it will be overloaded, bringing contention and less parallelism that possible. If it’s too deep, the counters at its leafs will not be fully utilized, achieving less than optimal throughput. Another sensitive parameter is the the prism width, i.e. the total number of prism locations in the balancers at a given level.
of the tree. This parameter affects the chances of a successful pairing-off. If the prism is too large then threads will tend to miss each other, failing to pair-off and causing contention on the toggle bit. If it’s too small, contention and sequential bottlenecking will occur as too many threads will be trying to access the same prism locations at the same time.

Thus using a diffracting tree to construct data structure make the data structure sensitive to the system load. therefore additional techniques must be used to force the algorithm to adopt itself to the current level of contention.

One such technique was suggested by Della-Libera and Shavit in the paper “Reactive diffracting trees” [20]. The reactive diffracting tree is a diffracting tree which can grow and shrink to better handle changing access patterns. In such tree, each node will act like a counter (i.e. when reached, the traversal of the tree stops even if this node is not a leaf) or a regular sub tree. The decision about each node is made periodically, according to current system load. In cases of high concurrency, all the nodes will be “unfolded”, allowing high distribution of the threads, in cases of low loads, the tree will “shrink”, sparing the high overhead of traversing through all the tree levels. The main drawback of such algorithm is that each node has to maintain its current state, which again, returns us to a centralized location in memory that everyone has to access, making it a hot-spot and reducing concurrency.
4.1. OVERVIEW

4.1.2 Elimination

Elimination is a state when two threads carrying a token and anti-token meet, and “eliminate” each other [73]. For example a producer thread can be defined as one carrying a token, accordingly a consumer is carrying an anti token. When such two threads meet in a data structure, they can exchange the information they carry.

A common way to use elimination is to build an elimination array. Similarly to the prism array presented in previous section, elimination array is a set of cells, where each thread randomly chooses a location and spins waiting another thread to “collide” with. When collision occurred, in case the two collided threads carry a token and anti-token, they can exchange information and leave the data structure.

In their paper about the construction of elimination trees [73], Shavit and Touito suggest to combine a diffracting tree with the elimination paradigm. If, while traversing the tree, a producer thread collides with a consumer thread in one of the prism arrays, the producer will hand out its item to the consumer and they both can stop their activity on the tree. Thus, threads that are of opposite types eliminate each other, without reaching the bottom of the tree. Elimination was used since then in construction of many concurrent data structures. Various types of concurrent queues and stacks combine elimination techniques.

In the paper about the construction of concurrent stacks [35] by Hendler, Shavit and Yerushalmi, elimination array is used in combination with a lock free stack to design a concurrent non-blocking, linearizable stack. A single elimination array is put in front of a lock free stack. Each thread that performs operation on the stack, first approaches the elimination array and tries to pair up with an opposite type thread. If a removing and an inserting thread meet, one takes the item of the other accordingly and they leave. Otherwise the thread continues to the lock free stack. This way a stack semantics is preserved, while introducing highly concurrent behavior at high loads. A similar technique was used later to construct a concurrent linearizable queue[61].

The main drawback of all the listed algorithms is that while they provide good performance in high load, in low loads the additional data structures like the elimination array or the elimination tree provide unnecessary overhead and have negative effect on the the performance. Additionally, in a manner similar to the diffracting trees, elimination algorithms use parameters that are sensitive to a system concurrency level. For best performance, the size of elimination array should be affected by the number of
threads concurrently accessing it. The length of the spin period, when one thread waits to pair-off with another, before it decides to move on is also a sensitive parameter, if it’s too small, it will cause contention, but if it’s too big threads will waste their time spinning when the chance to collision is small. Various techniques were suggested to handle those issues, such as backoff in time and space etc. We adopted some of those techniques to make our construction adaptive to a system load.

### 4.2 The ED-Tree

The idea behind the ED-Tree is combining the modified diffracting [76] tree as above with the elimination-tree techniques [73].

As a first step in constructing the ED-Tree we add to the diffracting tree a collection of lock-free queues at the output wires of the tree leaves as shown in Figure 4.3. Since the nodes of the tree will form a bottleneck if one uses the naive implementation of diffracting tree, we put prism arrays in each balancer as explained in Section 4.1.1. To perform an insert operation, producer threads traverse the balancers from the root to the leaves and then inserts the item onto the appropriate queue. In any quiescent state, when there are no threads in the tree, the output items are balanced out so that the top queues have at most one more element than the bottom ones, and there are no gaps. That way we get a structure where all requests end up properly distributed on the queues at the tree leaves, and the collection of all the queues together has the behavior of one large queue.

![Diagram of a Tree/4 tree](image)

**Figure 4.3: A Tree/4 leading to 4 lock-free queues.**

One can keep a second, identical tree for consumers, where the output wires of the tree reference the same queues as the first tree, and you will see that from one quiescent state to the next, the items removed are the first ones inserted onto the queue. We thus have a collection of queues that are accessed in parallel, yet act as one quiescent
4.2. **THE ED-TREE**

FIFO queue.

The second step is to merge the two trees to a single tree. To remove or insert an item a thread enters the balancer and tries to diffract using the prism array. If diffraction succeeds it moves to the next level, otherwise it approaches the toggle bit and advances to the next level.

Next we insert the elimination technique to the our construction. We use the observation, that if a producer and a consumer "collided" in a prism array any time during their tree traversal, there is no need for them to continue traversing the tree, as they can fulfill each other’s request. The producer will hand his item to the consumer, and each can leave the tree and return.

To achieve that, we replace each node of the tree with highly distributed nodes that use elimination and diffraction on randomly chosen array locations. Every prism in the tree is turned to "elimination prism", which allows not only diffraction, but also elimination, meaning that when two threads collide in a cell of such an array, they decide to eliminate or diffract according to the "partner’s" type.

We put an EDArray in front of the toggle bits in every balancer. If two removing threads meet in the array, they leave on opposite wires, without a need to touch the bit, as anyhow it would remain in its original state. If two inserting threads meet in the array, they leave on opposite wires at the same manner. If an insert or remove call does not manage to meet another in the array, it toggles the toggle bit and leaves accordingly. Finally, if a producer and a consumer meet, they eliminate, exchanging items. Threads that were "unlucky" to find an elimination partner all the way down the tree and reach the leaves, approach the queue referenced by the leaf. As we show later, in contended cases the percentage of such threads is very low, i.e, most of the threads eliminate successfully in the tree without reaching the queues at the output wires.

Our last modification to the tree, is adding an additional toggle bit to each node in the tree. That way each balancer will maintain an elimination prism array and two toggle bits, one for producer threads and one for consumer threads, thus allowing "inserting" and "removing" threads follow each other and not going to different directions if they weren’t diffracted. In this sense it differs from prior Elimination and/or Diffrac-tion balancer algorithms [73, 76] which had a single toggle bit instead of separate ones, and provided LIFO rather than FIFO like access through the bits. Figure 4.4 presents...
the final ED-tree construction. Each balancer in $Tree[4]$ is an elimination-diffraction balancer. The start state depicted is the same as in Figure 4.3, as seen in the producer’s toggle bits. From this state, an insert of item 6 by Thread A will not meet any others on the elimination-diffraction arrays, and so will toggle the bits and end up on the 2nd stack from the top. Two removes by Threads B and C will meet in the top balancer’s array, diffract to the sides, and end up going up and down without touching the bit, ending up popping the first two values values 1 and 2 from the top two lock-free queues. Thread F which did not manage to diffract or eliminate, will end up as desired on the 3rd queue, returning a value 3. Finally, Threads D and E will meet in the top array and “eliminate” each other, exchanging the value 7 and leaving the tree. This is our exception to the FIFO rule, to allow good performance at high loads, we allow threads with concurrent insert and remove requests to eliminate and leave, ignoring the otherwise FIFO order.

It can be shown that all insert and remove requests that do not eliminate each other provide a quiescently consistent FIFO queue behavior. Moreover, while the worst case time is $\log k$ where $k$ is the number of lock-free queues at the leaves, in contended cases, as our test show, $1/2$ the requests are eliminated in the first balancer, another $1/4$ in the second, $1/8$ on the third, and so on, which converges to an average of 2 steps to complete insert or remove, independent of $k$. 

Figure 4.4: Full $ED$-$Tree$. 

4.3 Implementation

4.3.1 The tree nodes

As described above each balancer (see the pseudo-code in Figure 4.5) is composed of an EliminationArray, a pair of toggle bits, and two references one to each of its child nodes. The last field, lastSlotRange (which has to do with the adaptive behavior of the elimination array), will be described later in Section 4.3.2.

```
1 public class Balancer{
2    ToggleBit producerToggle, consumerToggle;
3    Exchanger[] eliminationArray;
4    Balancer topChild, bottomChild;
5    ThreadLocal<Integer> lastSlotRange;
6 }
```

Figure 4.5: Balancer structure.

The implementation of a toggle bit as shown in Figure 4.6 is based on an AtomicBoolean which provides a compareAndSet API (CAS). To access it a thread fetches the current value (Line 5) and tries to atomically replace it with the complementary value (Line 6). In case of a failure the thread retries (Line 6).

```
1 AtomicBoolean toggle = new AtomicBoolean(true);
2 public boolean toggle(){
3    boolean result;
4    do{
5        result = toggle.get();
6    }while(!toggle.compareAndSet(result, !result));
7    return result;
8 }
```

Figure 4.6: Toggle bit.

The implementation of an eliminationArray is based on an array of Exchangers. Each exchanger maintains a place in memory where a thread can "declare" himself and spin waiting for another thread to arrive. When a thread enters the exchanger and sees that another thread is already waiting, the two threads exchange information, based on which each of them later decides whether to eliminate or diffract. Each exchanger (Figure 4.7) contains a single AtomicReference which is used as an Atomic placeholder for exchanging ExchangerPackage, where the ExchangerPackage is an object used to wrap the actual data and to mark its state and type. Each thread approaching ED-tree,
wraps its relevant data in a new instance of ExchangerPackage and the exchangers help threads to approach each others package in order to understand what type of thread was met and in case of elimination for consumer thread to reach the item that the producer thread tries to insert.

```java
public class Exchanger {
    AtomicReference<ExchangerPackage> slot;
}
```

```java
public class ExchangerPackage {
    Object value;
    State state; // WAITING/ELIMINATION/DIFFRACTION,
    Type type; // PRODUCER/CONSUMER
}
```

Figure 4.7: Exchanger structure.

Each thread performing either insert or remove traverses the tree as follows. Starting from the root balancer, the thread tries to exchange its package with a thread with an opposing operation, a producer tries to exchange with a consumer and vice versa. In each balancer, each thread chooses a random slot in the eliminationArray, enters the exchanger, publishes its package, and then backs off in time, waiting in a loop to be eliminated. In case of failure, a back-off in “space” is performed several times. The type of space back off depends on the cause of the failure: If a timeout is reached without meeting any other thread, a new slot is randomly chosen in a smaller range. However, if a timeout is reached after repeatedly failing in the CAS while trying to either pair or just to swap in, a new slot is randomly chosen in a larger range. If collision occurs, the thread examines it’s partner’s package and decides to diffract if the partner is of the same type and eliminate otherwise. The direction of the diffraction is determined by the order in which CAS operations occurred. Thread that succeeded to ”publish” itself first in the exchanger will go up and the other one will go down.

The result of the meeting of two threads in each balancer is one of the following four states: ELIMINATED, TOGGLE, DIFFRACTED0, or DIFFRACTED1. In case of ELIMINATED, a Consumer and a Producer successfully paired-up, and the method returns. If the result is TOGGLE, the thread failed to pair-up with any other type of request, so the toggle() method shown in Figure 4.5 is called, and according to its result the thread accesses one of the child balancers. Lastly, if the state is either
4.3. IMPLEMENTATION

4.3.2 Adaptivity

In the back off mechanism described above the thread senses the level of contention and depending on it selects randomly an appropriate range of the eliminationArray to work on (by iteratively backing off). However, each time a thread starts a new operation, it initializes the back-off parameters, wasting the same unsuccessful rounds of back-off in place until sensing the current level of contention. To avoid this, each thread saves its last used range between invocations (Figure 4.5 line 5). This saved range is used as (a good guess of) the initial range at the beginning of the next operation. Using this method proved to be a major factor in reducing the overhead in low contention situations and allowing the EDTree to yield good performance under high contention.

Additional adaptation mechanism we add to the backoff in space described earlier, is the following: If a thread failed to collide after certain amount of timeouts, implying that there are small amount of threads in the balancer and the load is low, it moves to spin on the first cell of the array for a certain amount of spins, before approaching the toggle bit and moving to the next level, that way increasing the chance of successful collision. Additionally a special flag is set to this thread indicating that in the next levels of the tree it should skip the elimination array and approach toggle bit immediately in order to reach the bottom of the tree as fast as possible without wasting unnecessary time, spinning and waiting to other threads. Since this type of scenario indicates that the contention in the tree is probably low, there will be no contention on the toggle bit, and moving to the next level will occur fast. Thus, the described mechanism helps indicating cases when the contention is low and the chance of meeting another thread in the tree is small, already at the first level of the tree, making threads avoid unnecessary overhead while traversing the rest of the tree.

As a final step, a thread that reaches one of the tree leaves, performs it’s action (insert or remove) on a queue. A queue can be one of the known queue implementations: a SynchronousQueue, a LinkedBlockingQueue, or a ConcurrentLinkedQueue.

Based on ED-Tree with the different queues we implemented the following three types of pools:
4.3.3 An Unfair Synchronous Pool [47]

When setting the leafs to hold an unfair $SynchronousQueue$, we get a $unfair\ synchronous\ pool$. As the synchronous queue, an unfair synchronous pool provides a “pairing up” function without the buffering. Producers and consumers wait for one another, rendezvous, and leave in pairs. Thus, though it has internal queues to handle temporary overflows of mismatched items, the unfair synchronous pool does not require any long-term internal storage capacity.

4.3.4 An Object Pool

With a simple replacement of the former $SynchronousQueue$ with a $LinkedBlockingQueue$, or a $ConcurrentLinkedQueue$ we get $Blocking\ Object\ Pool$, or a $NonBlocking\ Object\ Pool$ in respect. An $object\ pool$ is a software design pattern. It consists of a multi-set of initialized objects that are kept ready to use, rather than allocated and destroyed on demand. A client of the object pool will request an object from the pool and perform operations on the returned object. When the client finishes work on an object, it returns it to the pool rather than destroying it. Thus, it is a specific type of factory object.

Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low. The pooled object is obtained in predictable time when the creation of the new objects (especially over a network) may take variable time. In this work we show two versions of an object pool: blocking and a non blocking. The only difference between these pools is the behavior of the consumer thread when the pool is empty. While in the blocking version a consumer is forced to wait until an available resource is inserted back to Pool in the unblocking version a consumer thread can leave without an object.

An example of a widely used object pool is a $connection\ pool$. A connection pool is a cache of database connections maintained by the database so that the connections can be reused when the database receives new requests for data. Connection pools are used to enhance the performance of executing commands on the database. Opening and maintaining a database connection for each user, especially of requests made to a dynamic database-driven website application, is costly and wastes resources. In connection pooling, after a connection is created, it is placed in the pool and is used
4.4. PERFORMANCE EVALUATION

again so that a new connection does not have to be established. If all the connections are being used, a new connection is made and is added to the pool. Connection pooling also cuts down on the amount of time a user waits to establish a connection to the database.

4.3.5 Starvation Avoidance

Finally, in order to avoid starvation in our pool, we limit the time a thread can be blocked in the queues at the bottom of the tree before it retries the whole Tree\[k\] traversal again. Starvation (Though it has never been observed in all our tests) can take place when threads that reached the queues on the leaves wait too long for their request to be fulfilled, while most of the new threads entering pool eliminate on the way down the tree and leave fast.

4.4 Performance Evaluation

We evaluated the performance of our new algorithms on a Sun UltraSPARC\textsuperscript{TM} T2 Plus multicore machine. In all tests, the ED-tree we used is constructed from three levels, with 8 queues at the leaves. We found this construction optimal for the range of the contention levels that was tested (2 threads to 256 threads approaching in parallel), adding not too much overhead in case of low contention and providing a sufficient level of distribution in case of high contention.

We begin in Figure 4.8 by comparing the new unfair synchronous queue of Lea et al. [47] scheduled to be added to the \texttt{java.util.concurrent} library of JDK, to our ED-Tree based version of an unfair synchronous queue. The graph on the left is a zoom in of the low concurrency part of the one on the right. As we explained earlier, an \textit{unfair synchronous queue} provides a symmetric “pairing up” function without buffering: Producers and consumers wait for one another, rendezvous, and leave in pairs.

One can see that the ED-Tree version behaves similarly to the JDK version up to 8 threads (left figure). Above this number of threads the ED-Tree scales nicely while the JDK queue’s overall throughput declines. At its peak at 64 threads the ED-Tree delivers more than 10 times the performance of the JDK.

Beyond 64 threads the threads are no longer bound to a single CPU, and traffic across the interconnect causes a moderate performance decline for the ED-Tree version.
We next compare two versions of an object pool. An object pool is a set of initialized objects that are kept ready to use, rather than allocated and destroyed on demand. A consumer of the pool will request an object from the pool and perform operations on the returned object. When the consumer has finished with an object, it returns it to the pool, rather than destroying it. The object pool is a specific type of factory object. Unlike the unfair synchronous queue, the consumers wait in case there is an available object, while producers never wait for consumers, they add the object to the pool and leave.

We compared an ED-Tree BlockingQueue implementation to the LinkedBlockingQueue of JDK6.0. Comparison results for the object pool benchmark are shown on the lefthand side of Figure 4.9.

The results are pretty similar to those in the unfair synchronous queue. The JDK’s LinkedBlockingQueue performs better than its unfair synchronous queue, yet it still does not scale well beyond 4 threads. In contrast, our ED-Tree version scales well even up to 80 threads because of its underlying use of the LinkedBlockingQueue. At its peak at 64 threads it has 10 times the throughput of the JDK’s LinkedBlockingQueue.
Next, we evaluated implementations of concurrent queue, a more relaxed version of an object pool in which there is no requirement for the consumer to wait in case there is no object available in the Pool. We compared the java.util.concurrent.ConcurrentLinkedQueue (which in turn is based on Michael’s lock-free linked list algorithm [58]) to an ED-Tree based ConcurrentQueue. Again, the results show a similar pattern: the JDK’s ConcurrentLinkedQueue scales up to 14 threads, and then drops, while the ED-Tree based ConcurrentQueue scales well up to 64 threads. At its peak at 64 threads, it has 10 times the throughput of the JDK’s ConcurrentLinkedQueue.

Since the ED-Tree object pool behaves well at very high loads, we wanted to test how it behaves in scenarios where the working threads are not pounding the pool all the time. To this end we emulate varying work loads by adding a delay between accesses to the pool. We tested 64 threads with a different set of dummy delays due to work, varying it from 30-600ms. The comparison results in Figure 4.10 show that even as the load decreases the ED-Tree synchronous queue outperforms the JDK’s synchronous queue. This is due to the low overhead adaptive nature of the randomized mapping into the EDArray: as the load decreases, a thread tends to dynamically shrink the range of array locations into which it tries to map.

Another work scenario that was tested is the one when the majority of the pool users are consumers, i.e. the rate of inserting items to the pool is a lot lower than the one demanded by consumers and they have to wait until items become available. Figure 4.11 shows a workload in which 25% of the threads are producers, performing insert operations, and 75% of the threads are consumers, removing items from the pool. One can see, that while ED-pool is outperformed in low contention, it still scales nicely.
CHAPTER 4. SCALABLE PRODUCER-CONSUMER POOLS

Figure 4.11: Performance of Linked-Blocking queue and Synchronous queue.

and outperforms both JDK’s linked-blocking queue and synchronous queue in more contended workloads.

Figure 4.12 shows what happens when number of threads using the pool is steady, but the ratio of consumers changes from 50% to 90%, e.i from the scenario when number of producers and consumers is equal to one where there is a majority of consumers, meaning more remove that insert requests.

Then, investigated the internal behavior of the ED-Tree with respect to the number of threads, we check the elimination rate at each level of the tree. The results appear in Figure 4.13. Surprisingly, we found out that the higher the concurrency, that is, the more threads added, the more threads get all the way down the tree to the queues. At 4 threads, all the requests were eliminated at the top level, and throughout the concurrency range, even at 265 threads, 50% or more of the requests were eliminated at the at the top level of the tree, at least 25% at the next level, and at least 12.5% at the next. This, as we mentioned earlier, forms a sequence that converges to less than 2
4.5. CONCLUSIONS

Figure 4.13: Elimination rate by levels as concurrency increases.

Figure 4.14: Elimination range as the work load changes for 64 threads.

as \( n \), the number of threads, grows. In our particular 3-level ED-Tree tree the average is 1.375 balancer accesses per sequence, which explains the great overall performance.

Lastly, investigated how the adaptive method of choosing the elimination range behaves with different contention loads. Figure 4.4 shows that as we expected the algorithm dynamically adapts the working range according to the contention load. The more each thread spent doing work not related to the pool, the more the contention decrease and respectively the default range used by the threads decrease.

4.5 Conclusions

We investigated the widely used “resource pools”, in which allocated items are not destroyed but recycled after use in order to be used again. We showed that an ED-Tree structure used as “resource pools”, as our empirical testing shows, performs well at both high and low concurrency levels. These impressive results achieved thanks to the fact the ED-Tree does not have a central place through which all threads pass, and thus enables both parallelism and reduced contention. By building such data structure we showed that elimination and diffraction techniques can be combined to work well
at both high and low loads.
Chapter 5

Quasi-Linearizability
5.1 Overview

Linearizability \cite{42} is a useful and intuitive consistency correctness condition that is widely used to reason and prove common data structures implementations. Intuitively it requires each run to be equivalent in some sense to a serial run of the algorithm. This equivalence to some serial run imposes strong synchronization requirements that in many cases result in limited scalability and synchronization bottlenecks. In order to overcome this limitation, more relaxed consistency conditions have been introduced. Such alternative consistency conditions for concurrency programming include Sequential consistency\cite{52}, Quiescent consistency\cite{11}, Causal consistency\cite{6}, Release consistency\cite{29}, Eventual consistency\cite{84} and Timed consistency\cite{81}. But, the semantics of these relaxed conditions is less intuitive and the results are usually unexpected from a layman point of view. In this thesis we offer a relaxed version of linearizability that preserves some of the intuition, provides a flexible way to control the level of relaxation and supports the implementation of more concurrent and scalable data structures.

While Linearizability is the most intuitive condition (providing a serial perception), trying to maintain such strict definition often results in high contention bottlenecks. Despite these limitations Linearizability is today the most popular consistency model for high level data structures such as Map, Queue and Stack. The definition of linearizability is equivalent to the following:

- All function calls have a linearization point at some instant between their invocation and response.
- All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition.

This definition is intuitive and composable, making it useful in the implementations of concurrent linearizable Data-Structures from other linearizable objects. But, when using this model to write concurrent highly scalable data structures the results are often less than satisfactory in terms of performance and scalability.

For example, SEDA\cite{87}, the motivating and initiating reason for the current research is a common design pattern for highly concurrent servers, which heavily relies on thread pools. Such thread pools are composed from two elements (i) a set of threads ready to serve tasks and (ii) a task queue from which the threads consume their tasks. For
the task queue, state of the art concurrent queue of Michael and Scott\cite{58} is usually used. It is based on the fact that enqueue and dequeue may happen concurrently while threads trying to enqueue should race. Meaning such queue, which is not part of the server logic in a highly concurrent system, can become by itself a bottleneck limiting the overall SEDA system utilization. One can claim however, that more than often a thread pool does not need a strict FIFO queue, what is required is a queue with relaxed linearizability, in other words, a queue that does not allow one task to starve, meaning bypassed by more than a certain number of tasks.

Another common pattern is the shared counter, which in many applications may become a bottleneck by itself. In order to trim down this contention point Aspnes et al.\cite{11} offered a counting network which reduces the contention while maintaining a relaxed consistency condition called quiescent consistency. Such a relaxed counter can be used for example as an id generator, the output of this algorithm is a unique id for each requesting thread while a strict order is not required. This counter may also match other design patterns for example a “Statistical Counter.” Modern servers expose many statistical counters, mainly for administration and monitoring. These counters count “online” every operation done on the server. Due to their run time nature these counters by themselves may easily become a contention point. However, sometimes there is no real need for accurate numbers but to capture the general trend. On the other hand the main drawback of the counting network algorithm is also its relaxed consistency, such relaxation does not provide any upper bound for the “inconsistency.” We show in Section 5.6 that the upper bound is $N \times W$ where $N$ is the number of working threads, and $W$ is the width of the counting network.

Two more common examples for widely used data structures are the Hash Table and the Stack. While there is a broad range of highly concurrent implementations for a Hash Table as with the former examples the need for a linearizable implementation is often too strict. A very common use case for a Hash Table is a Web Cache. In this case a cache miss while the data is in the Cache might not be a desirable behavior but may be sacrificed for better scalability. More than that, even getting a stale data for a short while might not be a problem. A similar behavior commonly happens with a Stack, a linearizable LIFO implementation can ordinarily be replaced with an almost LIFO implementation for better scalability. Such implementation may return for a pop operation an element which is “close” to the top of the stack but not necessarily the
element at the top. In some usages, this weaker constraint may be acceptable for the gain of better scalability, for example consider a special form of resource pool where it is preferred to reuse the most recently used resource due to some higher cost of initialization of a resource which was not used recently. Such resource pool can use a stack to hold its resources and always take the resource from the top, or if replaced with the stack implementation mentioned above, it may take a resource which was used recently but not necessarily the last used resource.

The above examples have motivated us to provide a quantitative definition of the limited non-determinism that the application requirements might allow. We define a consistency condition which is a relaxed linearizability condition with an upper bound on the non-determinism. Each operation must be linearizable at most at some bounded distance from its strict linearization point. For example, tasks may be dequeued from a queue not in strict FIFO order but very close to it. That is, there is a linearization order in which every task \( t \) which is in the queue, may only be dequeued if there is no other task \( t' \) which is still in the queue such that \( t' \) has been originally enqueued into the queue before \( t \), and after \( t' \) was enqueued, \( k \) or more other tasks were enqueued prior to \( t \). This non strict FIFO order generates the desired behavior in which we can execute any of the first \( k \) tasks in the queue but we are not allowed to bypass the task at the front of the queue more than \( k \) times. Our definition is strong and flexible enough to define at the same time (continuing the above example) that a dequeue that returns empty may not be reordered, i.e., it has to be in its strict linearizable order.

In this thesis we introduce a formal definition of quasi-linearizability condition which captures this condition. This condition introduces some degree of non-determinism, but is useful to prove the quasi-linearizability of different implementations as exemplified in later sections.

5.1.1 Consistency Models

Many models were offered as weaker alternatives to Linearizability. Three of them are Quiescent consistency[11], Eventual consistency[84] and Timed consistency[81].

5.1.1.1 Quiescent consistency

This model provides high-performance at the expense of weaker constraints satisfied by the system. This property has three conditions:
(I) Operations should appear in some sequential order (legal for each object).

(II) Operations not separated by quiescent state may not occur in program order. E.g., A dequeue operation that overlaps two enqueue operations, enqueue x and then enqueue y, may return the second enqueued object - y.

(III) Operations whose occurrence is separated by a quiescent state should appear in the order of their occurrence and may not be reordered.

An object is in a quiescent state if currently there is no pending or executing operation on that object.

5.1.1.2 Eventual consistency

This model is a specific form of weak consistency; e.g., a storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. The most popular system that implements eventual consistency is the Internet DNS (Domain Name System). Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the last update.

These two models, in most cases, allow better concurrency but on the other hand do not provide any strict upper bound or an adaptive way to determine the “inconsistency” gap when compared to Linearizability. The following model does provide a special form of upper bound on the “inconsistency.”

5.1.1.3 Timed consistency

This model adds the notion of time to the occurrences of events and not just order, roughly speaking, timed consistency models require that if a write operation is executed at time $t$, it must be visible to all processes by time $t + \Delta$. As a result the actual implementations and guarantees they provide that satisfy either Timed consistency or quasi-linearizable conditions are very different. This model has some similarity to the quasi-linearizable model we provide, however, the concept of time is not equivalent to the concept of distance which we present in the quasi-linearizable model. Specifically the timed consistency model does not allow reordering of events. In the sequel we show implementations which are quasi-linearizable but not timed consistent. Another
difference is that timed consistency split operations into two groups, write or read. While quasi-linearizable separates operations according to their logical meaning.

5.2 The Definition of Quasi Linearizability

In this chapter we provide a formal definition for the Quasi-Linearizability condition for concurrent objects which is an extension to the Linearizability [42] condition. At first we start with a short review of the Linearizability condition definition, the definition follows the notations and standard model as in “The Art of Multiprocessor Programming” [41]. After the review, we will extend the condition definition into Quasi-Linearizability accompanied by various examples.

5.2.1 Linearizability Review

**Definition 1. History:** Following [42], a history is a list of events which are ordered according to the time line in which they occurred, each event represents either a method invocation or a method response, a method invocation event is represented by the tuple \(< O.\text{method}(\text{args}), T >\), where \(O\) is the object the invocation operates on, \(\text{method}\) is the invoked method, \(\text{args}\) are the invocation arguments and \(T\) is the thread that started this invocation. Method invocation response is represented by the tuple \(< O: t(\text{results}), T >\), where \(t\) is either \(\text{OK}\) or an exception name and \(\text{results}\) are the invocation result set. A response matches a prior invocation if it has the same object and thread, and no other events of \(T\) on object \(O\) appear between them.

**Definition 2.** History \(H\) is called sequential if the first event of \(H\) is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response.

For example, consider the following sequential history of a queue:

\[ H_1 = < \text{Queue.enq}(x), T_1 >, < \text{Queue.OK}(), T_1 >, < \text{Queue.enq}(y), T_2 >, < \text{Queue.OK}(), T_2 >, < \text{Queue.deq}(), T_1 >, < \text{Queue.OK}(x), T_1 > \]

And a non sequential (concurrent) history of a queue:

\[ H_2 = < \text{Queue.enq}(x), T_1 >, < \text{Queue.enq}(y), T_2 >, < \text{Queue.OK}(), T_1 >, < \text{Queue.OK}(), T_2 >, < \text{Queue.deq}(), T_1 >, < \text{Queue.OK}(x), T_1 > \]

A sequential history is considered legal if it is part of the sequential specification of the object, i.e it is a valid, single threaded sequential run of the object.
5.2. THE DEFINITION OF QUASI LINEARIZABILITY

Definition 3. An invocation is pending in \( H \) if no matching response follows the invocation.

For instance, in the history:
\[ H_3 = \langle \text{Queue.enq}(x), T_1 \rangle, \langle \text{Queue.enq}(y), T_2 \rangle, \langle \text{Queue.OK}(), T_1 \rangle, \langle \text{Queue.deq}(), T_1 \rangle, \langle \text{Queue.OK}(x), T_1 \rangle, \text{the invocation} \langle \text{Queue.enq}(y), T_2 \rangle \text{is pending in} \ H_3. \]

Definition 4. An extension of history \( H \) is a history constructed by appending zero or more responses matching the pending invocation of \( H \).

In the above example, the following two histories are an extension of \( H_3 \):
- \( H_3 \) itself by appending zero responses or \( H_3' = H_3, \langle \text{Queue.OK}() , T_2 \rangle \), by appending the missing response for the pending \( \langle \text{Queue.enq}(y), T_2 \rangle \) invocation.

Definition 5. \( \text{Complete}(H) \) is the sub-sequence of \( H \) consisting of all matching invocation and responses, thus, removing all pending invocations from \( H \).

In our case, \( \text{Complete}(H_3) = \langle \text{Queue.enq}(x), T_1 \rangle, \langle \text{Queue.OK}(), T_1 \rangle, \langle \text{Queue.deq}(), T_1 \rangle, \langle \text{Queue.OK}(x), T_1 \rangle \)

Definition 6. \( H|T \) is a history consisting of all and only the events of thread \( T \) in history \( H \),

\[ \bullet \ H_2|T_1 = \langle \text{Queue.enq}(x), T_1 \rangle, \langle \text{Queue.OK}(), T_1 \rangle, \langle \text{Queue.deq}(), T_1 \rangle, \langle \text{Queue.OK}(x), T_1 \rangle \]

\[ \bullet \ H_2|T_2 = \langle \text{Queue.enq}(y), T_2 \rangle, \langle \text{Queue.OK}() \rangle, \langle \text{Queue.OK}(y), T_2 \rangle \]

Definition 7. two histories \( H \) and \( H' \) are equivalent if for each thread \( T \), \( H|T = H'|T \).

We say that a method call \( m_0 \) precedes a method call \( m_1 \) in history \( H \) if \( m_0 \) finished before \( m_1 \) started: that is, \( m_0 \)'s response event occurs before \( m_1 \)'s invocation event.

Definition 8. Linearizability A history \( H \) is linearizable if it has an extension \( H' \) and there is a legal sequential history \( S \) such that:

(I) \( \text{Complete}(H') \) is equivalent to \( S \).

(II) If method call \( m_0 \) precedes method call \( m_1 \) in \( H \), then the same is true in \( S \).
5.2.2 Quasi-Linearizability

We first define, quasi-sequential specification and then define what a quasi-linearizable history is and finally define a quasi-linearizable data structure.

For example, consider the following sequential history of a queue:

\[ H = < \text{Queue.enq}(1), T_1 >, < \text{Queue.OK}(), T_1 >, < \text{Queue.enq}(2), T_2 >, < \text{Queue.OK}(), T_2 >, < \text{Queue.deq}(), T_1 >, < \text{Queue.OK}(2), T_2 >, < \text{Queue.enq}(3), T_1 >, < \text{Queue.OK}(), T_1 >, < \text{Queue.deq}(), T_1 >, < \text{Queue.OK}(1), T_1 >, < \text{Queue.deq}(), T_2 >, < \text{Queue.OK}(3), T_2 > \]

This sequential history is not legal for a queue, however, it is not “far” from being legal, by exchanging the order of enqueue of element 1 and 2, one can get a legal sequential history. To formally define this reordering of \( H \) and to express how “far” is \( H \) from a legal sequential history, we introduce the quasi-linearizable concept.

A sequential history is an alternating sequence of invocations and responses, starting with an invocation, and each response matches the preceding invocation. We substitute these two matching events by a single event:

\[ < O.method(args), t(results) > \]

while removing the executing thread indication. We can ignore the thread executing this method call in a sequential history because in a sequential history operations do not overlap and eventually we consider a sequential history legal if the order of events there can represent a single threaded execution. For example the above history \( H \) is written as follows in its sequential representation:

\[ H = < \text{Queue.enq}(1), \text{OK}() >, < \text{Queue.enq}(2), \text{OK}() >, < \text{Queue.deq}(), \text{OK}(2) >, < \text{Queue.enq}(3), \text{OK}() >, < \text{Queue.deq}(), \text{OK}(1) >, < \text{Queue.deq}(), \text{OK}(3) > \]

Unless specified otherwise all the sequential histories in the sequel are condensed in that way. Each event in such a history represents the tuple corresponding to both the invocation and the matching response.

A sequential specification is the set of all possible sequential runs of an object, each of these runs can be represented as a sequential history. The term legal sequential history specifies that a sequential history is part of the sequential specification of the object that generated that history.

**Definition 9.** For each event \( e \) in a sequential history \( H \), we define \( H[e] \) to be its index in the history, for two events, \( e \) and \( e' \), \( H[e'] < H[e] \) if and only if \( e' \) is before \( e \) in \( H \).
5.2. THE DEFINITION OF QUASI LINEARIZABILITY

- \( H_i \) is the \( i \)th element in \( H \). I.e., \( H[H[e]] = e \).
- \( \text{Events}(H) \) is the set of all the events in \( H \).
- \( \text{Distance}(H', H) \) the distance between two histories \( H \) and \( H' \), such that \( H' \) is a permutation of \( H \), is \( \max_{e \in \text{Events}(H)} \{|H'[e] - H[e]|\} \).

Notice that by definition, the distance is defined only for histories which are a permutation of each other.

**Definition 10. Object domain:** The set of all possible operations that are applicable to an object. We distinguish between operations that have different arguments or different return values. For example, for \( O = \text{stack} \), \( \text{Domain}(O) = \{< O.\text{push}(x), \text{void}>, < O.\text{pop}(), x \mid x \in X \} \cup \{< O.\text{pop}(), \phi >\} \), where \( X \) is the set of all the possible stack elements.

**Definition 11.** A sequential history \( H[D] \), is the projection of history \( H \) on a subset \( D \) of the events, i.e., \( H \) after removing from it all the events which are not in \( D \).

**Definition 12.** A sequential history \( H[O] \), is the projection of history \( H \) on an object \( O \) which is defined as \( H[O] = H|\text{Domain}(O) \).

We extend the sequential specification of an object \( O \) to a larger set that contains sequential histories which are not legal but are at a bounded “distance” from a legal sequential history. In other words, a sequential history \( H \) is in this set if there is some legal sequential history that its “distance” from \( H \) is bounded by some specified bound. We define that bound using a function that we name the Quasi-linearization factor. It is a function that operates on subsets of the object domain, mapping each subset to its “quasi factor”, which is the upper bound on the relative movement among the operations in the subset that turn it into a legal sequential history. Formally,

**Definition 13. Quasi-linearization factor:** A function \( Q_O \) of an object \( O \) defined as \( Q_O : D \rightarrow \mathbb{N} \). \( D \) is the set containing subsets of the object’s domain, formally \( D = \{d_1, d_2, \ldots \} \subset \text{Powerset(Domain}(O)) \).

- \( d_1, d_2, \ldots \) are not necessarily disjoint sets, The quasi factor for operations that do not appear in \( D \) is unbounded.
- The range of the function \( Q_O \) which is \( \mathbb{N} \) is extended to a more robust set in the sequel, but this suffices in order to grasp the general concept.
Definition 14. $Q_O$-Quasi-Sequential specification: is a set of all sequential histories that satisfy the “distance” bound implied by the quasi-linearization factor $Q_O$ of an object $O$. Formally, for each sequential history $H$ in the set, there is a legal sequential history $S$ of $O$ such that $H$ is a prefix of some history $H'$ which is a permutation of $S$ and $\forall$ subset $d_i \in D$: $\text{Distance}(H'|d_i, S|d_i) \leq Q_O(d_i)$

For example consider the previously mentioned history:

$H = \langle \text{Queue.enq}(1), \text{OK}() \rangle, \langle \text{Queue.enq}(2), \text{OK}() \rangle, \langle \text{Queue.deq}(), \text{OK}(2) \rangle, \langle \text{Queue.enq}(3), \text{OK}() \rangle, \langle \text{Queue.deq}(), \text{OK}(1) \rangle, \langle \text{Queue.deq}(), \text{OK}(3) \rangle$

And the following Quasi-linearization factor for the queue object ($Q$) in the history $H$:

- $D_{\text{enq}} = \{ \langle \text{Queue.enq}(x), \text{OK}() \rangle \mid x \in X \}$
- $D_{\text{deq}} = \{ \langle \text{Queue.deq}(), \text{OK}(x) \rangle \mid x \in X \}$
- $Q_{\text{Queue}}(D_{\text{enq}}) = 2$
- $Q_{\text{Queue}}(D_{\text{deq}}) = 0$

There exists a legal sequential history

$S = \langle \text{Queue.enq}(2), \text{OK}() \rangle, \langle \text{Queue.enq}(1), \text{OK}() \rangle, \langle \text{Queue.deq}(), \text{OK}(2) \rangle, \langle \text{Queue.enq}(3), \text{OK}() \rangle, \langle \text{Queue.deq}(), \text{OK}(1) \rangle, \langle \text{Queue.deq}(), \text{OK}(3) \rangle$

such that for $\text{Distance}(H|D_{\text{enq}}, S|D_{\text{enq}}) \leq 2$ (in our case the distance is exactly 1), and $\text{Distance}(H|D_{\text{deq}}, S|D_{\text{deq}}) = 0$. From the above we get that $H$ is part of the $Q_Q$-Quasi-Sequential specification because in this case we say that $H$ is a prefix of the same history $H$ (it self), and $H$ is a permutation of $S$ and it satisfies the distance bounds implied by the quasi factor $Q_{\text{Queue}}$.

Definition 15. Let $\text{Objects}(H)$ be the set of all the objects that $H$ involves with.

Definition 16. Q-Quasi-Linearizable history: A history $H$ is $Q$-Quasi-Linearizable if it has an extension $H'$ and there is a sequential history $S'$ such that:

(I) $Q = \bigcup_{O \in \text{Objects}(H)} Q_O$. $Q$ is a union of all the different object quasi-linearizable factors, each object has its own separate domain even for objects of the same type.

(II) $\text{Complete}(H')$ is equivalent to $S'$. 
5.2. THE DEFINITION OF QUASI LINEARIZABILITY

(III) If method call $m_0$ precedes method call $m_1$ in $H$, then the same is true in $S'$.

(IV) $\forall O \in Objects(H) : S'|O$ is member of the $Q_O$-Quasi-Sequential specification.

We notice that a linearizable history $H$ has $Q$-quasi-linearizable factor 0 for all of the domains of the objects that appear in it, i.e., for each object $O$ in $H$, $Q(Domain(O)) = 0$.

Definition 17. Q-quasi-linearizable object: An object implementation $A$ is Quasi-Linearizable with $Q$ if for every history $H$ of $A$ (not necessarily sequential), $H$ is $Q$-Quasi-Linearizable history of that object.

For example, consider the following quasi-linearization factor for a blocking queue implementation which is $Q_{Queue}$-Quasi-Linearizable:

$D_{enq} = \{< Queue.enq(x), void > | x \in X \}$, $D_{deq} = \{< Queue.deq(), x > | x \in X \}$

Domain(Counter) = $D_{enq} \cup D_{deq}$

- $Q_{Queue}(D_{enq}) = k$
- $Q_{Queue}(D_{deq}) = 0$

Practically it means that an enqueue operation can bypass at most $k$ preceding enqueue operations (and an arbitrary number of dequeue operations occurring in between). This quasi-linearizable queue specifications may be used as the task queue in the SEDA[87] system described in the Introduction.

5.2.3 Definition details

5.2.3.1 H is a prefix of some history?

Consider the following history for a concurrent counter:

$H = < Counter.getAndInc(), 3 >, < Counter.getAndInc(), 1 >$ This history can never be reordered to a legal sequential history since the event $< Counter.getAndInc(), 2 >$ is missing. However, it is reasonable for an execution of a quasi-linearizable implementation of a counter to create such a history because the execution can be stopped at any time. By appending the missing response and invocation $< Counter.getAndInc(), 2 >$ at the end of $H$ we can reorder this history to a legal sequential history. This addition of unseen future events is described in the definition by adding a sequential history $H$ to the quasi-sequential specification of the counter object if it is a prefix of some
history which that history is equivalent to a legal sequential history, the some history is $H \cup < \text{Counter.getAndInc()}, 2 >$. If we do not allow completion of unseen events, hence do not place $H$ in the quasi-sequential specification of the counter, we reduce the definition strength since any implementation would have been forced to return the entire range of getAndInc() results for not yet terminated operations (without skipping any numbers as the return value) in order to be quasi-linearizable, which in fact makes it similar to quiescent consistent, for instance, a single thread operating on the object has to get a fully linearizable contract from the implementation. It is important to notice that by adding non existing future events, it is not possible to make any history quasi-linearizable. For instance, if the quasi factor for the getAndInc() operation is 5, the following history $H = < \text{Counter.getAndInc()}, 8 >$ can never be transformed to a legal sequential history only by adding any future events, that is because no matter what unseen future events are added, the first event needs to be moved at least by distance 7 in a legal sequential history (because there are 7 events that must occur before it in any legal sequential history).

5.2.3.2 Distance measured on each subset of the domain separately

The distance is measured only on the projection of the entire history on a subset of the domain, this is done intentionally since some operations may have no effect on others and we do not want to take them into account when we calculate the distance, For instance, consider the following history (written in a not formal form):

$H = \text{enq}(1), \text{size}()=1, \text{size}()=1, \ldots, \text{size}()=1, \text{enq}(2), \text{deq}()=2, \text{deq}()=1.$

If we measure the distance on the enqueue operation and consider the $\text{size}()=1$ operations between $\text{enq}(1)$ and $\text{enq}(2)$, then the distance is unbounded, since an unbounded number of $\text{size}$ operations may be executed (in this case one should consider a subset containing all possible $\text{enq}$ operations separately). Another notion is that the subsets of the domain that has a quasi factor are not necessarily disjoint, which can be used to define a more generic quasi state. For instance it may be interesting to disallow reordering between $\text{size}$ and $\text{enqueue}$ operations, but to allow a reorder between $\text{enqueue}$ and $\text{dequeue}$ operations.
5.2.3.3 Extend Quasi Linearizable Factor Bound

In the definition we have specified that the bound for each domain subset is a constant number, however, in some cases (as shown later on the Bitonic Counting Network), the bound can vary depending on different parameters such as configurable implementation parameters or different use cases (i.e., the number of threads accessing the object concurrently or different system properties). This is addressed by providing the ability to specify a custom function as the bound instead of a constant bound, and that function arguments take the parameters mentioned above. Formally, instead of having $Q_O : D \rightarrow \mathbb{N}$ we change the function as follows: $Q_O : D \rightarrow F^N$ where $F^N$ is the set of all functions into $\mathbb{N}$. Formally, $F^N = \{ f | f$ is a function and $\text{Range}(f) = \mathbb{N} \}$. This way, a function $f$ that represents a bound of a domain subset can receive the above variables as its parameters.

5.2.3.4 Predicting the future?

Consider the following history for a concurrent queue:

$H = \text{enq}(1), \text{enq}(2), \text{deq}()=3, \text{enq}(3), \text{deq}()=1, \text{deq}()=2.$

By the definition of quasi-linearizability, this history is quasi-linearizable for $Q(D_{\text{enq}}) \leq 2$, however, it may seem weird that we consider this history legal because the first dequeue operation returns an element which has not yet been enqueued. However, practically speaking, if there was an implementation of a concurrent queue that this history represents an execution of it, it would mean that the implementation is predicting the future, which obviously is not feasible. The only type of such implementation would be one that returns a random value on its dequeue operation. However, for a data structure implementation to satisfy quasi-linearizability, all of its possible execution histories must be quasi-linearizable and given an implementation that returns a random result, we can easily schedule one execution example which may never be transformed to a legal sequential history while keeping the quasi distance boundaries.

5.2.3.5 Locality (Composition)

Following [42], a property P of a concurrent system is said to be local if the system as a whole satisfies P whenever each individual object satisfies P. As shown in [42], linearizability is a local property, that is a history $H$ is linearizable if and only if, $\forall O \in \text{Objects}(H) : H|O$ is linearizable.
Theorem 18. $H$ is $Q$-Quasi-Linearizable if and only if, $\forall O \in Objects(H) : H|O$ is $Q_O$-quasi linearizable.

Proof: It is easy to see that if the entire history $H$ is quasi linearizable, then the projection of it on each object is quasi-linearizable by definition. In order to prove the other direction we need to show that given a history $H$, such that $\forall O \in Objects(H) : H|O$ is $Q_O$-quasi-linearizable, $H$ is $Q$-quasi-linearizable. For each object $O$, we denote $H'_O$ as the extension of $H|O$ implied by the quasi-linearizable definition of $H|O$ and $S'_O$ as the sequential history that it is part of the $Q_O$-Quasi-Sequential specification of $O$ such that $Complete(H'_O)$ is equivalent to $S'_O$. By definition, $S'_O$ is a prefix of some history which is a permutation of a legal sequential history, we denote that legal sequential history by $S_O$. We define $H' = H \bigcup_{O\in Objects(H)} H'_O$, clearly $H'$ is an extension of $H$. We construct $S$ by replacing all of the objects sub-histories $Complete(H'|O)$ with $S'_O$, clearly $S$ is equivalent to $Complete(H')$ and the order of method invocations is kept between the two histories. We need to show that $\forall O \in Objects(H) : S|O$ is part of the $Q_O$-Quasi-Sequential specification of $O$, we get the above since $S|O = S'_O$ by construction. QED

Composition is important in order to be able to use Quasi-Linearizable objects in a bigger system while keeping the Quasi-Linearizable property of the entire system. For instance, consider a system that keeps track of some internal components and operations and at some point needs to calculate the total number of operations executed on the system. Normally, such a system uses a linearizable shared counter that counts each of the operations occurrences, and a combined display counter that represents the total number of operations that is calculated by summing up all operation counters. Assume we have 2 counters for 2 different operations, we get the total number of operations by adding this two counters, assume this counters have a $k_1$, and $k_2$ respectively constant quasi-linearizable bounds for their add method. From the composition derives that the quasi bound for the combined counter is $k_1 + k_2$ since the bound is kept for each counter upon composition. (If $k_1 = k_2 = 0$ we get a fully linearizable combined counter).

5.2.3.6 Operations that are invisible

The definition treats all operation as equal, however, operations that are invisible (i.e, do not change the state of the object) can pose difficulties on actual implementations if they affect the distance equally as visible operation since such implementations proba-
bly need to update some internal state for each of these operation in order to comply with the distance bounds. For instance, consider a queue that supports 3 operations: enqueue, dequeue and size, size in this case is considered as an invisible operation. There are a few natural ways to define such a quasi-linearizable queue, one would be to put the enqueue and size operations in the same domain subset in the quasi-linearization factor specification, as well as the dequeue and size operations, thus disabling the queue to return the wrong size value at any stage. However, these boundaries take size operations into consideration when calculating the distance of reordered enqueue operations. An alternative would be to put size in a separate domain subset, however, this results in legal quasi-linearizable implementations that return size that was legal at some state. Intuitively, the distance between two visible operations should not be affected by invisible operation executed between the two. On the other hand, there is still a need for a bound on the reordering distance of invisible operation, otherwise one cannot pose any limitations for this type of operations in a quasi-linearizable object. In order to address this, we can extend the Distance of two histories $H$ and $H'$ in the following way:

- Let $VEvent(H)$ be all the events that appear in $H$ that are visible.
- Let $IEvent(H)$ be all the events that appear in $H$ that are invisible.
- $Event(H) = VEvent(H) \cup IEvent(H)$
- $VDistance(H, H') = \max_{e \in VEvents(H)} \{|H'|VEvents(H)[e] - H[VEvents(H)[e]]\}$
- $NDistance(H, H') = \max_{e \in IEvents(H)} \{|H'[e] - H[e]|\}$
- $Distance(H, H') = \max\{NDistance(H, H'), VDistance(H, H')\}$

Using this upgraded distance definition, the enqueue and size operations can be placed together in the same subset and also the dequeue and size operations, while we consider size to be an invisible operation.
5.3 Implementations

5.3.1 Random Dequeued Queue

5.3.1.1 Implementation

We offer a simple quasi linearizable non blocking queue implementation, given in Algorithms 3 and 4, and illustrated in Figure 5.1.

The idea of the implementation is to spread the contention of the dequeue method by allowing to dequeue an element which is not at the head of the queue, but not more than \( k \) places away from the head. We base our quasi queue implementation on [58] which is based on a linked list, in fact our enqueue operation is exactly the same. We change the dequeue operation to pick a random index between 0 and \( k \) (the quasi factor), if the picked index is larger than 0 it iterates over the list from the head to the item at the specified index, it attempts to dequeue it by doing a single CAS(compare and set) which attempts to mark it as deleted, If it succeeds, this is the dequeued item. If failed it reselects a random index and retries the entire process. The algorithm will retry the process a few times up to a pre-configured max retries count, eventually if the process did not succeed to dequeue a random element after all the retries, it falls back to the scenario as if index 0 is picked. If the selected index is 0, the operation iterates over the list from the head until it finds a node which has not yet been dequeued. While iterating it attempts to remove all encountered dequeued nodes by attempting to advance the head of the list using a CAS each step.

![Figure 5.1: Random Dequeue Queue](image)

This implementation retains an “almost” FIFO behavior by allowing the following weaker constraints: Dequeue operation may return results not in the precise order they were enqueued, but not too “far” from it (up to a constant bound). When an empty
5.3. IMPLEMENTATIONS

(null) dequeue result is returned, there are no remaining enqueued items in the queue. Formally, we describe this behavior with the following quasi-linearizable factor:

- $Q_{nb\rightarrow queue}(D_{enq}) = k$ (we may reorder enqueue operations up to distance $k$, which dictates the actual dequeue order).

- $Q_{nb\rightarrow queue}(D_{deq} \cup \{< deq(), null >\}) = 0$ (no additional reordering of dequeue operations is allowed).

- $\forall x \in X : Q_{nb\rightarrow queue}(\{< enq(x), void >, < deq(), null >\}) = 0$ (enqueue operation can not be reordered if it goes over an empty dequeue operation)

Algorithm 3: Enqueue: Quasi linearizable queue

```
input: x : value
1 // Allocate new node with the enqueued value
2 // Node structure {value:object, next:pointer, deleted:boolean}
3 newNode ← new Node();
4 newNode.value ← x;
5 while true do
6     // tail : A node pointing to the tail of the queue
7     lastNode ← tail;
8     nextNode ← lastNode.next;
9     // Check that tail and lastNode are still consistent
10    if lastNode = tail then
11        // Was tail pointing to the last node?
12        if nextNode =⊥ then
13            // Try to link the node at the end of the list
14            if CAS(lastNode.next, nextNode, newNode) then
15                // Enqueue is complete, try to
16                // swing tail to the inserted node
17                CAS(tail, lastNode, newNode);
18                break;
19            end
20        else
21            // Tail was not pointing to the last node,
22            // try to swing it to the next node
23            CAS(tail, lastNode, nextNode);
24        end
25    end
26 end
```
Algorithm 4: Dequeue: Quasi linearizable queue

```plaintext
retries ← 0;
while true do
    // head : A node pointing to the head of the queue
    first ← head, last ← tail, next ← first.next;
    // Check that first and head are still consistent
    if first = head then
        if first = last then
            if next = ⊥ then
                return ⊥;
            else
                // Tail is lagging behind, help it
                CAS(tail, last, next);
            end
        else
            randomIndex ← 0;
            if retries < MAX_RETRIES then
                // K : Predefined quasi factor
                randomIndex ← Random(K), retries ← retries + 1;
                if randomIndex > 0 then
                    value = AttemptRandomDequeue(randomIndex, next);
                    if value ≠ ⊥ then
                        return value;
                    end
                // if failed repeat entire process
            end
        end
    else
        randomIndex ← 0;
        if retries < MAX_RETRIES then
            // K : Predefined quasi factor
            randomIndex ← Random(K), retries ← retries + 1;
            if randomIndex > 0 then
                value = AttemptRandomDequeue(randomIndex, next);
                if value ≠ ⊥ then
                    return value;
                end
            // if failed repeat entire process
        end
    end
    // iterate from head until encountered undeleted node
    while next ≠ ⊥ and next.deleted = true do
        // Try to remove node by advancing head
        if first ≠ last and CAS(head, first, next) then
            first ← next, next ← next.next;
        end
        // Queue is empty
        if next = ⊥ then
            return ⊥;
        end
        // Attempt to dequeue first element
        if CAS(next.deleted, false, true) then
            return next.value;
        end
    end
end
```

Algorithm 5: AttemptRandomDequeue: Quasi linearizable queue

\begin{algorithm}
\textbf{input:} randomIndex : Integer, next : Node
1 \hspace{1em} i \leftarrow 0;
2 \hspace{1em} \text{// Move up to selected random index}
3 \hspace{1em} \textbf{while} \hspace{0.5em} i < \text{randomIndex} \hspace{0.5em} \textbf{do}
4 \hspace{1.5em} \text{if} \hspace{0.5em} \text{next.next} = \perp \hspace{0.5em} \text{then}
5 \hspace{2em} \text{break;}
6 \hspace{2em} \textbf{end}
7 \hspace{1.5em} \text{next} \leftarrow \text{next.next}, \text{i} \leftarrow \text{i} + 1;
8 \hspace{1em} \textbf{end}
9 \hspace{1em} \text{// Try to dequeue node at random index}
10 \hspace{1em} \textbf{if} \hspace{0.5em} \text{CAS}(\text{next.deleted}, \text{false}, \text{true}) \hspace{0.5em} \text{then}
11 \hspace{1.5em} \text{return} \hspace{0.5em} \text{next.value;}
12 \hspace{1em} \text{else}
13 \hspace{1.5em} \text{return} \hspace{0.5em} \perp;
14 \hspace{1em} \textbf{end}
\end{algorithm}

5.3.1.2 Proof of Quasi-Linearizability

In this part we show that the random dequeued queue implementation satisfies \(Q_{\text{nb\_queue}}\)-Quasi Linearizability using the formal definition. Roughly speaking, we use a similar mechanism as in regular linearizability of selecting linearization point in the algorithm that determines the sequential full order between the overlapping and not overlapping method invocations. In this case we name these as quasi-linearization points. In full detail, we take a run of the algorithm and treat this run history as \(H\), we need to show that the history \(H\) is \(Q_{\text{nb\_queue}}\)-Quasi Linearizable. According to the definition we need to extend \(H\) to \(H'\) by adding responses for some pending method invocations. In order to do so we need to decide which pending methods invocation took place and which did not. We do that by selecting quasi-linearization points that consists of location and condition in the algorithm. We add a missing response in \(H'\) for a pending method only if its execution path have reached or passed a corresponding quasi-linearization point and met its condition. \(\text{Complete}(H')\) consists only of non-pending methods of \(H\) and the pending methods that became non-pending by adding the missing responses in the process described in the previous. The quasi-linearization points of the algorithm are as follows:

Enqueue method (Algorithm 3):

- line 14 and the condition is a succesfully executed CAS which corresponds to adding the node at the end of the list.
Dequeue method (Algorithm 4):

- line 1 and the condition is that the method returns an empty queue state (line 7).

- line 11 and the condition is a successfully executed CAS which corresponds to dequeuing the element that was marked as deleted by the CAS operation.

- line 28 and the condition is that next is NULL which corresponds to an empty queue result (line 33)

- line 37 and the condition is a successfully executed CAS which corresponds to dequeuing the first element that was marked as deleted by the CAS operation.

After constructing Complete($H'$) we need to show a sequential history $S'$ that is equivalent to $Complete(H')$. We take each method invocation and its response in $Complete(H')$ and use the corresponding linearization point to mark a single point in time that method affect takes place. Doing so we specify a full order between the methods and create a sequential history $S'$. $S'$ is equivalent to $Complete(H')$ because it maintain the order of events per thread and for each method call $m_0$ that precedes method call $m_1$ in $Complete(H')$ the same is true in $S'$ because the quasi-linearization points are inside each method invocation response time interval and the full order derived by them cannot change order of non overlapping methods. In our case we have one object in the history, a $nb-queue$, thus $S'|nb-queue = S'$. We need to show that $S'$ is a member of the $Q_{nb-queue}$-Quasi-sequential specification

We recall that a history is in the $Q_{nb-queue}$-Quasi-sequential specification if it is a prefix of some sequential history which is a permutation of a legal sequential history of a non blocking queue, and the distance between that history and the legal sequential history satisfies the quasi-linearization factor. In our case we take $S'$ it self to be the “some history” such that $S'$ is a prefix of (every history is a prefix of it self), and show that $S'$ is a permutation of some legal sequential history $S$ such that $Distance(S'|D_{enq}, S|D_{enq}) \leq k$. From the algorithm, the enqueue order which is specified at $S'$ according to the linearization points, also dictates the order of nodes that the linked list of elements is constructed of. If our implementation does not satisfy the distance for the enqueue operations, then for each legal sequential history $S$, $Distance(S'|D_{enq}, S|D_{enq}) > k$. This means that we can not reorder enqueue operation
in any way in order to make it a legal sequential history, such that there is no enqueue
operation that moved over \( k \) other enqueue operation during the reorder process of \( S' \)
to a legal sequential history \( S \). Since dequeue order is dictated by the enqueue order,
the above is equivalent to that there is at least one dequeue operation in \( S' \) that re-
turns an element which is placed inside the queue at a depth of at least \( k + 1 \) (there
are at least \( k \) elements before it including the head which are closer to the head of the
queue). That means that the dequeue operation needs to traverse over at least \( k \) not
deleted nodes until it reaches that element which it have dequeued. That contradicts
the algorithm since if it picks 0 as a random index, in that case it returns the first not
deleted node. Or if it picks an index between 1 and \( k - 1 \), it iterates over that number
of nodes from the head regardless of their deleted state, it could never reach a node at
a distance of \( k \) from the head.

For simplicity we denote \( D_{deq^+} = D_{deq} \cup \{< deq(), null >\} \). In a non blocking queue
sequential specification, the order of dequeue operation results is determined by the
order of enqueue operation, hence, it is enough to reorder just the enqueue operation
in order to make it a legal sequential history. Therefore, we can transform \( S' \) to a
legal sequential history \( S \) by just reordering the enqueue operation without reordering
any dequeue operations and get the required distance for dequeue operations, hence
\( Distance(S'|D_{deq^+}, S|D_{deq^+}) = 0. \)

We can see from the algorithm that an empty dequeue result is returned in both
cases only when there was a point in time during the dequeue operation that there
were no nodes in the list, that is the head was pointing to the tail and the next
element was null. When this case happens, from the order of operations derived by the
quasi-linearization points, prior to that event in \( S' \), there must be the same number of
enqueue operations and dequeue operations that returned a non null result, otherwise
there must be a node that is not deleted (dequeued). Therefore, there is no need to
move any enqueue operations over a null dequeue operation in the reordering of \( S' \) to
a legal sequential history. That satisfies the last bound, which is
\[
\forall x \in X : Distance(S'|\{< enq(x), void >, < deq(), null >\}, S|\{< enq(x), void >, < deq(), null >\}) = 0.
\]
5.4 Segmented Queue

5.4.1 Implementation

The previous implementation of a quasi linearizable queue is quite simple and intuitive but it only reduces contention on the dequeue operation while the enqueue threads still compete over the tail reference when trying to enqueue new elements. Additionally, a dequeue operation iterates over its randomly selected number of nodes, while it may traverse over a node that it can dequeue along the way. That implementation is quite simple and easy to learn from as an introduction example to the quasi-linearizable concept, moreover it fit to some scenarios, such as allocating tasks or ids from a previously filled pool of tasks and ids that were inserted to the queue in a not contentioned environment. The dequeuers in these scenarios are competing with each other to get tasks or ids. However, in a more general scenario, such as the SEDA [87], the enqueue and dequeue are working in parallel and the previous implementation still have high contention on the tail. In the following section we present an implementation given in Algorithms 6 and 8, and illustrated in Figure 5.2. This implementation scatters the contention both for dequeue and enqueue operations and in the normal case, iterates over less nodes while still keeping a constant quasi factor.

The general idea is that the queue maintains a linked list of segments, each segment is an array of nodes in the size of the quasi factor (specified by \texttt{quasiFactor+1} in the implementation details), and each node has a deleted Boolean marker, which indicates if it has been dequeued. Each enqueue iterates over the last segment in the linked list in some random permutation order; When it finds an empty cell it performs a CAS operation attempting to enqueue its new element, if failed it repeats this process. In case the entire segment has been scanned and no available cell is found (implying that the segment is full), then it attempts to add a new segment to the list and repeat the previous process.

The dequeue operation is similar. The dequeue iterates over the first segment in the linked list in some random permutation order. When it finds an item which has not yet been dequeued, it performs a CAS on its deleted marker in order to “delete” it, if succeeded this item is considered dequeued, if failed it repeat this process. In case the entire segment was scanned and all the nodes have already been dequeued (implying that the segment is empty), then it attempts to remove this segment from the linked
5.4. SEGMENTED QUEUE

list and repeats the process on the next segment. If there is no next segment, the queue is considered empty.

Figure 5.2: Segmented Queue

Based on the fact that most of the time threads do not add or remove segments, most of the work is done in parallel on different cells in the segments. This ensures a controlled contention depending on the segment size, which is the quasi factor.

Data types:
Node {value: data type, next: pointer, deleted: boolean}
Segment {nodes: array of size \( k \) of Node’s, next: pointer, deleted: boolean }
Permutation {a permutation of the numbers 1 to \( k \)}

Shared variables:
tail : A segment pointing to the tail segment of the queue
head : A segment pointing to the head segment of the queue
\( K \) : Predefined quasi factor

Figure 5.3: Segmented queue definitions

The method \( \text{CreateLastSegment}(\text{expectedLastSegment}) \) (Algorithm 7) creates and adds a new last segment only if the current last segment is the provided method argument (expectedLastSegment). The result of the method is the current last segment which was either created by this invocation or another concurrent invocation if the current last segment is different than the method argument.

The method \( \text{GetNewSegment} \) that appears part of the \( \text{CreateLastSegment} \) method (Algorithm 7) can try to reuse segments from a pool that were previously generated using the \( \text{CreateLastSegment} \) but were not attached to the segments list due to failed
Algorithm 6: Enqueue: Segmented Quasi linearizable queue

\textbf{input}: $x$ : value

1 // Allocate new node with the enqueued value
2 newNode ← new Node();
3 newNode.value ← x;
4 // Get last segment
5 lastSegment ← tail;
6 if lastSegment = ⊥ then
7  lastSegment ← CreateLastSegment(⊥)
8 end
9 while true do
10  // Iterate over the segment at a random permutation
11  permutation ← GetRandomPermutation();
12  for $i$ ← 1 to $K$ do
13    randomIndex ← permutation.Get($i$);
14    if lastSegment.Get(randomIndex) = ⊥ then
15      // Found empty cell, try to enqueue here
16      if CAS(lastSegment.Get(randomIndex), ⊥, newNode) then
17        return;
18      end
19    end
20  end
21 // If reached here, no empty position, create a new segment
22 lastSegment ← CreateLastSegment(lastSegment);
23 end
Algorithm 7: CreateLastSegment: Segmented Quasi linearizable queue

\textbf{input}: \textit{expectedLastSegment} : Segment

\textbf{while true do}
\begin{algorithmic}[1]
\State lastSegment ← tail;
\State nextSegment ← lastSegment.next;
\State // Check if the expected last match,
\State // otherwise a new segment was already added
\If {lastSegment = \textit{expectedLastSegment}}
\State // Make sure lastSegment and tail still agree
\If {lastSegment = tail}
\State // Check if next segment after last segment do not exist
\If {nextSegment = \perp}
\State // Prepare a new segment and attempt to
\State // insert it as the last segment
\State newSegment ← GetNewSegment();
\State \If {CAS(lastSegment.next, NULL, newSegment)}
\State return newSegment;
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\State // Return the current last segment which
\State // was added by another thread
\State return tail;
\EndWhile
\end{algorithmic}
CAS at line 65. In fact, we have done so in our java implementation that is used to benchmark this algorithm.

The method \texttt{RemoveFirstSegment(expectedFirstSegment)} removes the first segment only if the current first segment is the provided method argument (expectedFirstSegment). The result of the method is the current first segment.

### 5.4.2 Proof of Quasi-Linearizability

This queue implementation satisfies the previously defined $Q_{nb\text{-}queue}$-quasi linearizable properties. The general idea is that a dequeue operation is always done on the first segment until all the elements of that segment have been dequeued, the same goes for the enqueue operation, each segment is fully filled before a new segment is created.

The quasi-linearization points of the algorithm are as follows:

- **Enqueue method (Algorithm 6):**
  - line 19 and the condition is a successfully executed CAS which corresponds to inserting the node into the segment and enqueing the value.

- **Dequeue method (Algorithm 8):**
  - line 8 and the condition is the firstSegment is NULL which corresponds to an empty queue result (line 3).
  - line 18 and the condition is the entire segment was scanned and there was at least one NULL node and all the non-NULL nodes were marked as deleted. This in turns correspond to an empty queue result (line 32) if the double scan of NULL nodes shows no difference in the NULL nodes state.
  - line 13 and the condition is a successfully executed CAS which corresponds to marking a node as deleted and dequeue its value.

One can see that in order to create a legal sequential history from the sequential history derived by the quasi-linearization points after extending and completing it in a similar way of the previous implementation proof, the order of the enqueue operation in the derived sequential history would need to be reordered only with the corresponding dequeue events that belong to the same segment only, without crossing into other segments dequeue operation.
Algorithm 8: Dequeue: Segmented Quasi linearizable queue

1. firstSegment ← head;
2. while true do
3.     nullIndexes ← {};
4.     // Queue is empty
5.     if firstSegment = ⊥ then
6.         return ⊥
7.     end
8.     // Iterate over the segment at a random permutation
9.     permutation ← GetRandomPermutation();
10.    for i ← 1 to k do
11.       randomIndex ← permutation.Get(i);
12.       node ← firstSegment.Get(randomIndex);
13.       // Found a candidate for dequeue, try to delete it
14.       if (node ≠ ⊥) then
15.           if CAS(node.deleted, false, true) then
16.               return node.value;
17.           end
18.       else
19.           // Keep track of null nodes in order get
20.           // a snapshot of null nodes if needed
21.           nullIndexes.Add(randomIndex);
22.       end
23.     end
24.     // Scanned the entire segment without finding
25.     // a candidate to dequeue If there were empty
26.     // cells, check if their state has changed
27.     if nullIndexes ≠ {} then
28.         foreach i in nullIndexes do
29.             // An element was enqueued to one of the previously
30.             // empty nodes, retry dequeue cycle
31.             if firstSegment.Get(i) ≠ ⊥ then
32.                 break;
33.         end
34.     end
35.     // Null indexes didn’t change, we have
36.     // a legit snapshot, queue is empty
37.     return ⊥
38. else
39.     // All nodes have been dequeued, we can safely remove
40.     // the first segment and retry dequeue
41.     firstSegment ← RemoveFirstSegment(firstSegment);
42. end
43. end
Algorithm 9: RemoveFirstSegment: Segmented Quasi linearizable queue

while true do
    firstSegment ← head;
    tailSegment ← tail;
    next ← firstSegment.next;
    // Check for consistent variable state
    if firstSegment = head then
        // Check for empty queue state
        if firstSegment = tailSegment then
            // Check if next segment after first segment do not exist
            if nextSegment = ⊥ then
                return ⊥;
            else
                // Tail is lagging behind, help it
                CAS(tail, tailSegment, next);
            end
        end
        result ← next;
        // If the first segment is the expected first
        // segment, try to remove it
        if value = expectedFirstSegment then
            CAS(head, firstSegment, next);
            // We can successfully break loop because the expected
            // first segment was removed by this thread or another
            break;
        end
    end
    // Return the current first element
    return head;
This assumption relies on the fact that according to the algorithm and quasi-linearization selected points, the corresponding enqueue events have all occurred either before other segments enqueue events or after other segments enqueue events without interleaving with other segments enqueue events. Since each segment length is of the quasi factor \((k)\), this means that during the reordering of the derived sequential history in order to make it legal sequential history, the enqueue events in that history that belongs to the same segment, moves at most \(k\) places, as required.

There are two possible algorithm paths of the dequeue operation that returns an empty queue result. The first one at line 3, with the corresponding quasi-linearization point at line 8, may only occur when there is no pending elements to dequeue otherwise the first segment could have not been null since the linearization point of the enqueue operation is only after there is at least one segment in the segments list. The second one at line 32, which we linearize at line 18, indicates that there was no available element to dequeue at that point because after iterating over the entire segment there was no undeleted node and only nodes without enqueues element. The double scan of the NULL nodes guarantee a snapshot of the NULL nodes states meaning that no new values were enqueued to the segment between the first iteration and the second.

### 5.4.3 Segmented Stack

The above algorithm can be adapted very easily in order to implement a non-blocking stack instead of a non-blocking queue. The difference is that when a push operation needs to add a new segment, it adds it to the head of the list of segments instead of the tail, thus having \textit{CreateFirstSegment} instead of the \textit{CreateLastSegment} method. And the pop operations is similar to the dequeue operation. Obviously this yields higher contention when adding and removing segments because both are competing on the same node in the segment list but most operations are still spread across a segment.

### 5.5 Performance Evaluation

We evaluated the performance of our new algorithms implemented in Java\textsuperscript{TM} on a Sun UltraSPARC\textsuperscript{TM} T2 Plus multicore machine. The benchmark consist of even number of enqueuers and dequeuers threads, each thread executing the same number of operations thus having 50\% enqueue and 50\% dequeue operations.
We can see from Figure 5.4 that the Segmented queue implementation outperforms both Michael and Scott\cite{58} and Random Dequeue when the number of threads increases, which is reasonable since it spreads both enqueue and dequeue contention. However as we increase the quasi factor, the overhead of scanning an entire segment just to realize the enqueuer needs to create a new segment or the dequeuer needs to remove the first segment, is increasing. On the other hand the Random Dequeue behaves very similar to the Michael and Scott algorithm when the quasi factor is low, but on high number of threads it improves if we increase the quasi factor, which is because the contention is reduced on the dequeue operation.

### 5.6 Bitonic[W] Counting Network

So far we have demonstrated the quasi-linearizable definition for queues and shown two algorithms of non-blocking queues that satisfies quasi-linearizability. In this part we show the definition is quite generic and it fits to other data structures as well, in this case a shared counter. Moreover, we show that the already existing Bitonic Counting Network\cite{11} which is an implementation of a counter, satisfies quasi-linearizability. Formally, A Bitonic[W] Counting Network (\(W\) the network width) is \(Q_{\text{counter}} - \text{quasi} -\)
linearizable, $Q_{\text{counter}}(D_{\text{inc}}) = \{< \text{getAndInc()}, n > | n \in \mathbb{N}\} \leq N \times W$ (where $N$ is the number of working threads). For example for the Bitonic[4] Counting Network showed in Figure 5.5 with $N = 4$ we show that $Q(D_{\text{inc}}) \leq 16$.

Figure 5.5: Bitonic[4] Counting Network

For the Bitonic Counting Network we choose the quasi linearization points when a thread is increasing the counter at the exit from the network, we denote this sequential history derived by the quasi linearization points as $S'$ (The process of taking a concurrent history and making a sequential history out of it using the quasi-linearization point is described in details in Chapter 5.3 at section 5.3.1.2).

**Lemma 19.** For any $n$, in $S'$ all the operations $\{< \text{getAndInc()}, m > | m \leq n - (N \times W)\}$ precede $< \text{getAndInc()}, n >$.

**Proof:** Assume in contradiction that the lemma does not hold, denote by $< \text{getAndInc()}, m >, m < n - (N \times W)$ as the missing operation. From the quiescent consistency property of Bitonic network, we know that if we now schedule an execution that lets all the threads that are currently traversing the network (executing getAndInc operation) to finish their current operation, and prevent new threads or the same threads to reenter the network, the network must be at a legal state, that is, all the values up to $n$ have been returned. From the algorithm of the network, we know that if $< \text{getAndInc()}, m >$ has not been executed yet, so does $\forall i > 0 : < \text{getAndInc()}, m + W \times i >$ (because this operations are diverted to the same counter). From that we get that $< \text{getAndInc()}, m >, < \text{getAndInc()}, m + W >, \ldots, < \text{getAndInc()}, m + W \times (N - 1) >$ have not happened by the time that $< \text{getAndInc()}, n >$ occurred. Since there are at most $N - 1$ threads that are pending execution completion, they can never fill in the missing $N$ operations to close the gap, in contradiction to the quiescent consistency property of the Bitonic network.

From the lemma we know that each getAndInc() operation had bypassed at most
$N \times W$ other getAndInc() operation, therefore we can find a legal sequential history $S$ which satisfies $Distance(S'|D_{inc}, S|D_{inc}) \leq N \times W$. In this part we have shown an upper bound of $N \times W$ for getAndInc operation, but this is not necessarily a tight bound.

### 5.7 Conclusions

We have shown a more relaxed concurrent model for linearizability and a few actual implementations which take advantage of the new model and are more concurrent than the equivalent linearizable implementation. We have demonstrated this with a queue that supports enqueue and dequeue operations, we can see how this definition can be adapted to a queue that also supports a peek operation. One way to define its quasi-linearizability, is by specifying the quasi factor parameters as follows: $Q(D_{enq} \cup D_{peek}) = k$ and $\forall x : Q(\{<\text{deq}(), x >, <\text{peek}(), x >\})=0$, meaning that a dequeue and peek operations may return an item at distance $k$ from the head of the queue, but a peek can not return an item which has been already dequeued. Additionally we have shown that the already known Bitonic counting network implementation is quasi linearizable. This model can be applied to specify other quasi linearizable objects, such as, stack, heap etc., and thus allows a more concurrent implementation of these objects.
Bibliography


[21] Brian Demsky and Alokika Dash. Evaluating contention management using discrete event simulation. In Fifth ACM SIGPLAN Workshop on Transactional Computing, April 2010. 3.4.2, 3.4.2.2

[22] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In In Proceedings of the 20th International Symposium on Distributed Computing (DISC’06), pages 194–208, September 2006. 1.2.2, 1.2.4, 2.3, 2.3.1, 2.4.2, 3.3.1.1


[40] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, NY, USA, 2008. 1.3.1

[41] Maurice Herlihy and Nir Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2008. 5.2


[49] Jens Knoop, Oliver Ruther, and Bernhard Steffen. Lazy code motion. SIGPLAN Not., 27(7):224–234, 1992. 1.2.3, 3.3.2.2


actional applications for multi-processing. In Proceedings of IISWC, September
2008. 2.4.2, 2.4.2.2

[60] M. Moir. Practical implementations of non-blocking synchronization primitives. In
Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed
Computing, August 1997. 1.2.2

[61] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to
implement scalable and lock-free fifo queues. In SPAA ’05: Proceedings of the sev-
enteenth annual ACM symposium on Parallelism in algorithms and architectures,
pages 253–262, New York, NY, USA, 2005. ACM. 1.3.1, 4.1.2

Inc., San Francisco, CA, USA, 2000. 1.1

[63] E. Morel and C. Renvoise. Global optimization by suppression of partial redund-

[64] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits,
James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey
Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and imple-
mentation of transactional constructs for C/C++. In OOPSLA ’08: Proceedings of the 23rd annual ACM SIGPLAN conference on Object-oriented programming
ystems, languages and applications, pages 195–212, 2008. 1.2.2, 3.1


[66] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. ACM Transac-
tions on Database Systems, 33(6):668–676, 1990. 2.4.2.1

ACM, 33(6):668–676, 1990. 2.1.1

[68] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional mem-
ory. In ISCA ’05: Proceedings of the 32nd Annual International Symposium on
Computer Architecture, pages 494–505, Washington, DC, USA, 2005. IEEE Com-
puter Society. 1.2.1

In Proc. of the International Symposium on Distributed Computing (DISC),
pages 284–298, Sep 2006. 1.2.2, 2.1, 2.3, 2.3.2

[70] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional mem-
ory with scalable time bases. In SPAA ’07: Proceedings of the nineteenth annual
ACM symposium on Parallel algorithms and architectures, pages 221–228, New
York, NY, USA, 2007. ACM. 2.4.2

[71] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java
using annotated constraints. pages 43–55. ACM Press, 2001. 3.2.2.2


[78] Byoungro So and Mary W. Hall. Increasing the applicability of scalar replacement. In CC, pages 185–201, 2004. 1.2.3


[80] Raheem Syed, Nahil Sobh, Umberto Ravaioli, Gabriel Popescu, and Mohamed Mohamed. imagej, May 2009. 1.2.2


