Composable Operations on High-Performance Concurrent Collections

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Nathan G. Bronson
October 2011

Abstract

Consider the task of designing the API for a data structure or reusable software component. There is a tension between simplicity and completeness: a simple interface is easier to learn and implement, but a complex interface is likely to include more of the functionality desired by the end user. The fundamental tool for managing this tension is composition.

Composition allows complex operations to be built up from a few primitive ones. Much of the art of API design comes in choosing a set of primitives that is both simple and complete. Unfortunately, existing efficient concurrent data structures don't have the ability to safely compose operations. The lack of a generic mechanism for composition leads their APIs to be both more complex and less complete.

This thesis presents new algorithms and techniques that allow composability for thread-safe sets and maps, while retaining excellent performance and scalability. First, we use optimistic techniques inspired by software transactional memory (STM) to add lazy copy-on-write to an efficient concurrent binary tree. This enables the collection to provide a linearizable `clone` operation whose running time is independent of the size of the tree. Second, we introduce transactional predication, a technique that allows STM integration of a concurrent map while bypassing the STM for most memory accesses. This improves the performance and scalability of the resulting transactional map, making it more suitable as a replacement for existing non-composable thread-safe maps. Finally, we explore the coexistence of lock-free algorithms and STM, producing an efficient transactional hash trie that includes lazy copy-on-write. We find that by tailoring the data structure for the STM environment and separately considering the case of non-transactional accesses, we provide both full composability and excellent performance and scalability.

Acknowledgements

Firstly, I am indebted to my wife Paola Grasso Bronson for her patience and support. She will probably be the only epidemiologist to read this thesis completely.

I would like to thank my advisor Kunle Olukotun. He has always challenged me to focus on the bigger picture, and he has always taken time to guide me even when he was busy. Thanks also to Darlene Hadding for her rock-solid administrative support.

I would like to thank Christos Kozyrakis and Dawson Engler for serving on my orals and reading committees, and thank Alex Aiken and Chuck Eesley for serving on my orals committee. Christos provided me with much hands on feedback when I was learning how to do research. Dawson's CS240 seems destined to be my favorite class ever. Special thanks to Alex for organizing a long string of interesting talks at the software lunch.

I would like to thank Woongki Baek, Brian Carlstrom, Jared Casper, Hassan Chafi, Jae-Woong Chung, Sunpack Hong, Austen McDonald, Chi Cao Minh, Tayo Oguntebi, Martin Trautmann, and the rest of the TCC group that initially attracted me to Stanford. Special thanks to Brian and Leslie Wu for helping me transition from professional programming back to school, and to JaeWoong for including me in my first conference paper push.

I would like to thank Anand Atreya, Kevin Brown, HyoukJoong Lee, Sang Kyun Kim, Lawrence McAfee, and Arvind Sujeeth, who joined me with some other TCC stragglers to become Kunle's students in the PPL. The cultural, technical and moral discussions (sometimes all three at once) at our weekly lunch with Kunle were always stimulating.

Special thanks to Hassan, who was instrumental to the research in this thesis and to my ability to complete it. I hope that I was as useful and as understanding an office mate for him as he was for me.

Thanks to Jacob Leverich for his generous help with cluster and LDAP wrangling.

Thanks to Mike Bauer and Megan Wachs for impressing me with their work ethic and fairness as we shared coarse assistant loads.

Martin Odersky's team at EPFL have produced something amazing in Scala. It is a testament to their great work that carefully engineered Scala data structures can be much more compact and beautiful than Java ones, yet compete head-to-head on performance. Thanks especially to Aleksandar Prokopec for his detailed and insightful feedback on my work.

I would like to thank Mooly Sagiv for pressing me to try to formalize my intuitions, and Juan Tamayo, Guy Gueta and Ohad Shacham for letting me be part of an exciting collaboration with Mooly and Alex. Thanks to Peter Hawkins, who seems to understand every idea, no matter how complicated, after at most one discussion.

I would like to thank Harold Reiter, who opened many doors for me when I was in high school. It would be difficult to overstate the effect that his intervention has had on my education. Thanks to John Board, my undergraduate advisor, who helped lay a foundation for my doctoral work long before I even considered graduate school.

Finally, thanks to my parents Karen and Dick, who instilled in me a confidence in my ability to reason and to learn. When I asked questions as a child their response was often, "Let's think about that." The wide-ranging discussions that followed are a template for all of the problem solving that I've done since. I would also like to specifically thank my dad for insisting that I learn Pascal instead of Basic.

Contents

List of Figures

Chapter 1

Introduction

Modern software systems are enormously complicated; that complexity is only possible due to the separation of software into components or abstractions. If the interface provided by a component is simpler than its encapsulated details, then it reduces the complexity that must be tackled at any one time. The ideal abstraction is one that takes responsibility for complex behavior while presenting a simple API. Completeness and simplicity are in tension, however, since adding functionality to a component eventually requires increasing the size of its interface.

Composition is the fundamental tool for managing the simplicity–completeness trade-off. Rather than providing a method or entry point for every potential use case, a composable API provides a small set of primitive operations that are composed to cover more complex scenarios.

In imperative programming, the primary mechanism for composition is sequencing. To compose two operations on a mutable object or data structure, merely call them one after the other. For example, when designing the API of Java's *`Map`* interface there was no need to provide a function to change the key associated with a value (`rename(`k_0`,`k_1`)`). The user can easily implement this functionality by following $v =$`remove(`k_0`)` with `put(`k_1`,`v`)`.

Unfortunately, composition by sequencing doesn't work for shared state in multi-threaded programs. Without some form of concurrency control there is no guarantee that a thread's operations won't get interleaved with another's. The lack of generic composability is a large problem when designing the API of a concurrent data structure, because we can no

longer rely on sequencing to cover all use cases from a small set of primitives.

1.1 The Parallelism Explosion – Multi-Core

Until recently, the lack of composability of concurrent data structures was less important. Parallel programming was restricted to a few domains in which the increased complexity and expense was justified by the total computational power required. Most CPUs had only a single thread of control, most computers had only a single CPU, and most programs were only written to take advantage of a single processor at a time. Parallel programs are more expensive to develop, and the cost was difficult to justify in an era during which the sequential performance of processors grew exponentially.

The enormous gains in microprocessor speed during the past several decades were driven by higher clock frequencies, larger caches, and more aggressive instruction-level parallelism (ILP). Power constraints have stopped frequency scaling, and most applications have reached a point of diminishing returns from increases in cache size or ILP logic complexity. Moore's law still describes the growth of available transistors, but these transistors can no longer be efficiently translated into sequential performance. All major processor vendors have switched to multi-core strategies, in which the bulk of the continuing performance gains will be available only for explicitly parallel programs. Herb Sutter succinctly describes this transition: "The free lunch is over" [86].

Parallelism is the simultaneous execution of multiple instructions or operations, which involves multiple control contexts. Multi-core processors explicitly expose these contexts as threads accessing a shared memory. Parallel programming models provide an abstraction that maps the execution of a program to resources of the underlying hardware.

By allowing more processors to be applied to a single task, parallelism improves performance. Better performance makes larger tasks feasible and reduces the time needed to complete existing tasks. Parallelization can also be used to reduce the power needed to perform a computation within a fixed time window.

Once the sequential programming model has been abandoned, parallelism can be used to improve power efficiency by moving to a larger numbers of cores where each core is slower. Lower clock frequencies yield a super-linear reduction in required power. Cores

with a lower throughput also have less need for the inter-instruction speculation currently used to achieve peak single-threaded performance, which means that they can avoid wasting power on incorrect predictions. The Sun Niagara processor is perhaps the best example of this approach that retains the instruction set of a high-performance sequential processor [52]. GPUs take this trend further by providing an enormous number of threads that each operate in a restricted environment.

1.2 Surveying Solutions for Parallel Programming

The set of application domains that can benefit from parallelism is very large, as is the set of parallel programming models. Each parallel programming model and environment imposes a set of rules or restrictions on the communication and synchronization of the execution contexts. When the model fits the application domain, these rules make it simpler to reason about the results of an execution. For example, in the producer–consumer model we can immediately ignore the consumer when looking for a bug in the producer, because the producer is completely isolated from the consumer.

It is the general opinion of the author that an application should use the most restrictive parallel programming model that yields an efficient solution, where efficiency includes execution costs and software engineering costs. The extra structure will allow more up-front analysis, which can be used to improve scheduling, arrange data to reduce the need for communication, and perform less synchronization. Pre-built structure moves functionality into code that is more widely reused, which leverages the engineering costs applied to that code. Models that guarantee deterministic output dramatically ease the application of the testing techniques developed for sequential programs.

Restrictive parallel programming models don't simplify reasoning if they are not a good fit for the problem domain. If many actors must coordinate to complete a task, for example, then the messages that are exchanged must include many pieces of intermediate state that would be more naturally expressed as local variables in a threaded program. To paraphrase Einstein's Razor,[1] everything should be as restrictive as possible, but no more.

[1] Everything should be made as simple as possible, but no simpler.

To motivate our contribution we will attempt to make a coarse map of parallel programming models and relevant application characteristics. While shared memory multithreading is one of the least restrictive of the models, it is both widely used and useful as a generic substrate for models with more structure.

1.2.1 Structured communication

If concurrently executing contexts are isolated from one another, then variations in their scheduling do not change a computation's result. Many of the difficulties in parallel programming stem from non-determinism, so we will first examine some systems that automatically provide deterministic outputs under parallel execution. We note that the term deterministic execution is too strong; many sources of non-determinism exist even for single-threaded executions[2].

Independent parallelism

Many parallel problems can be partitioned into pieces of work that involve no shared state or synchronization, except to aggregate their output. These are sometimes referred to as "embarrassingly parallel" or "trivially parallel." While these cases may be embarrassing or trivial for researchers studying parallelism, they are a best case scenario for the user. Independent parallelism for batch jobs is supported directly by job queuing systems like Torque, or may be as simple as arranging for a shell script to launch several copies of a program in the background. For systems that respond to external requests independent parallelism is often referred to as sharding, and uses a load balancing layer to randomly or deterministically distribute requests.

Relational queries

Concurrent execution involves multiple control contexts, but these do not have to be exposed to the programmer. SQL queries, for example, are expressed as a mathematical relation over a set of mutable data. The query's definition does not include the notion of an instruction pointer, so there are many execution strategies that can compute the results of

[2]Thanks to Doug Lea for pointing this out.

the relation. In practice databases may use multiple threads, processes or nodes to compute a result. Microsoft's LINQ system provides a similar way to express in-memory queries that are independent of control context, and that can be executed sequentially or in parallel [61].

Functional languages

Relational queries can be executed in parallel because their result is defined independently of their execution order. Functional languages provide this guarantee for their pure subset, which often accounts for the bulk of computationally intensive work. Pure computations can be run in parallel with no changes to a program's semantics, because referential integrity guarantees that each statement will produce the same answer regardless of when it is executed. Data Parallel Haskell is a recent system targeted at parallel execution of functional programs [51]. It provides convenient bulk pure operations on nested vectors using compilation techniques originally pioneered by NESL [8]. Parallel execution of these vector operations is an implementation detail, and it is not reflected in the semantics.

Producer–consumer parallelism

Producer–consumer parallelism arranges a set of isolated sequential processes into a directed acyclic graph, where the outputs from one process are inputs to others. Unlike more general forms of message passing communication, this structure can be rephrased as a functional computation over infinite lazy streams, and therefore produces the same answer on every execution. Unix pipes are perhaps the most widely used implementation of producer–consumer parallelism. This form of parallelism is also used in a fixed form inside the rendering pipeline of a modern GPU, or more flexibly in the GRAMPS programming model [84].

Replicated updates

Read-only operations in a request processing system can be considered pure functions, so they can be executed in parallel with each other. Read-only operations can be concurrent with updates if the updates are isolated, or if the reads can be satisfied from a snapshot

or replica of mutable state. This can be implemented in a shared memory system using immutable data structures or data structures that support lazy copy-on-write (such as those presented in later chapters of this thesis). In a multi-node system, updates can be propagated from a master to replicas by log-shipping or with a replicated finite state machine. A distributed consensus protocol can be used to elect a master in a fault-tolerant fashion.

Sequential semantics with parallel execution

Imperative languages may take advantage of parallel execution while presenting a single instruction pointer to the user. This can be done opportunistically for individual operations on compound data, as when a single call to the BLAS API executes a large matrix operation using parallel threads.

Automatic parallelization of fine-grained sequential operations is a holy grail of static analysis and compiler research. Historically Fortran has been the language of choice for this work, due to its limited aliasing and wide use in numerical computations. Unfortunately, this approach is hindered by the incidental dependencies and memory reuse prevalent in an imperative language without garbage collection. If the programmer or a static analysis can deduce that an operation should be repeated across contiguous memory elements then vector instructions allow for more efficient execution than explicit loops.

As an alternative to static analysis of intra-thread dependencies, architectural researchers have suggested speculative techniques that execute code in parallel while providing sequential semantics. Out-of-order execution that provides instruction level parallelism (ILP) is present in most desktop, laptop and server processors. Thread level speculation (TLS) proposals substantially extend the boundary at which speculation is performed and checked, typically speculating at loop boundaries in numeric workloads [38]. So far no commercial processor has been released that contains TLS support. Sun's cancelled Rock processor incorporated the similar technique of scout threads. Scout threads never commit their work, but serve instead to preload the L1 cache so that the main thread will encounter fewer memory stalls [88].

Dynamic resolution of dependencies and anti-dependencies can also be done in software, so long as enough work is done during each operation to amortize the dynamic costs. Stanford's DELITE toolkit enables the creation of Domain Specific Languages (DSLs) that

provide parallel execution while presenting sequential semantics to the user [21]. This system takes advantage of the Scala language's support for Language Virtualization to perform some scheduling work at compile time [20].

Another possibility is to restrict an imperative sequential language so that parallel execution is always possible. The Liszt domain-specific language for partial differential equations prohibits accesses to shared data that cannot be statically analyzed. Liszt users perceive that they are working within a restricted sequential language, but the restrictions allow the compiler to generate efficient parallel code operating over an unstructured mesh.

Isolated explicit parallelism

Concurrent contexts do not need to be hidden in order to be isolated from each other. The Sequoia programming language, for example, explicitly models threads of control on each node of a parallel machine [28]. The compiler statically prohibits these threads from communicating with any concurrent contexts; communication occurs with the parent only when the child is not running. (Recent extensions to Sequoia relax this restriction, allowing some message passing patterns to be expressed [5].)

Isolation can also be enforced by convention. The Hadoop implementation of Google's MapReduce model does not statically prevent node instances from observing their identity or their position in the network, or from maintaining state across invocations [22]. So long as this state is transient and reproducible, however, (such as memoized computations) the MapReduce model will still provide scalable execution of a deterministic calculation. MapReduce and Hadoop explicitly tolerate node failures by persisting the inputs and outputs of subcomputations.

1.2.2 Partitioned address spaces

One way that we can impose structure on inter-context communication is by allowing arbitrary communication, but requiring that it be explicit.

Message passing

The largest supercomputers are programmed almost exclusively using explicit message passing, with the MPI standard being the most common. In this environment the underlying processors have partitioned address spaces, and the explicit sends and receives of MPI correspond to the inter-node communication primitives implemented by the networking hardware. Point-to-point and collective communication are available. Originally MPI communications required the sender and receiver to rendezvous; the MPI-2 extensions add one-sided operations. While the individual processes in an MPI system are typically sequential, the overall system is not automatically repeatable or free of logic races. MPI focuses on execution efficiency, rather than programmability.

Erlang is a programming language that integrates the model of communicating sequential processes. Erlang processes are both units of execution and of isolation, and it is typical to have many processes inside each VM. Isolation between processes is enforced by the language. Unlike MPI, the default behavior of messages is asynchronous delivery via named queues. Erlang's original focus was on reliable distributed systems, but it has recently been applied also to situations whose primary focus is scalability.

Actors

If communicating sequential processes are actually run inside a shared address space then immutable data may be shared between sender and receiver as an optimization. The Scala programming language provides an actor library that allows threads to communicate with typed messages by passing references to shared immutable data within a single virtual machine [36]. While this does not provide the same scalability as distributed message passing systems, the overheads of messages can be substantially lower. Systems that use actors with a shared address space can also benefit from incremental use of direct access to shared mutable state for efficiency or convenience purposes.

1.2.3 Structured scheduling

The proliferation of multi-core processors has dramatically increased the number of systems which have the capability to run multiple threads or processes with shared memory.

Whereas true parallel execution was once limited to high-end servers with multiple processor chips, it now includes virtually all commodity microprocessors and extends even to embedded devices such as smart phones.

If multiple parallel contexts of a shared-memory architecture read and write the same memory location, then they have communicated. One way of structuring this communication is to explicitly enumerate the parallel contexts that may be active at any point in time. This explicit set of concurrent contexts can be used to reason about potential memory access interleavings and the resulting implicit communication.

Declarative loop-level parallelism

OpenMP provides declarative parallelism, with a focus on parallel execution of loops. The user is responsible for identifying or avoiding conflicting memory accesses; data races are not automatically detected. OpenMP performs implicit barrier synchronization at many locations, as well as providing explicit synchronization operations. Static analysis can allow OpenMP programs to be compiled for systems that do not implement shared memory in hardware.

Fork-join

Cilk++ adds language primitives for forking and joining lightweight threads to C++, and executes them using an efficient work-stealing runtime. It is a commercially supported descendent of the Cilk research language [9]. Like OpenMP, Cilk++ gives threads enough structure so that data races can typically be avoided statically and without locks. The fork-join model handles irregular and imbalanced parallelism better than the loop-centric model of OpenMP, but it is intimately bound to shared memory hardware.

Phased execution using barriers

While OpenMP and Cilk++ bound parallel threads using statically scoped boundaries, shared memory programs can dynamically impose a high degree of regularity on their execution using barriers. A barrier serves as a rendezvous between all of the members of

a group, guaranteeing that all execution in a phase before the barrier precedes any execution in the following phase. Recently the `Phaser` abstraction has been introduced in the Fortress language and the Java concurrency libraries; this subsumes the functionality of barriers while allowing group membership to vary dynamically [82].

1.2.4 Conflict avoidance or repair

In the most general case, threads can be created and destroyed arbitrarily; when reasoning about a particular algorithm or operation in this case one must assume that any conflicting operation might occur. This model provides safety by detecting potential conflicts and reactively avoiding them.

Locks and condition variables

Mutual exclusion locks provide a strong ordering between threads, but with much less structure than barriers. It is known that if two threads both protect access to some data by a lock, then one of them must have completed the protected access before the other started. It is not known ahead of time, however, which thread will go first. This means that unless updates are restricted to associative functions (or some other application-specific knowledge is applied) locks can produce non-deterministic execution.

Effective use of locks requires a protocol that guarantees that conflicting operations hold the same lock and a protocol that prevents deadlock. Locks can be associated either with data or with the code that accesses the data; the former typically scales better. Condition variables cooperate with locks to allow threads to efficiently wait for arbitrary conditions.

Atomic instructions

All modern multi-threaded architectures provide either the atomic compare-and-swap (CAS) or load-linked store-conditional (LL-SC). When combined with a retry loop, these allow a single memory location to be atomically sampled and transformed. These instructions form the foundation for implementing higher-level synchronization operations. They can also be used directly to construct concurrent data structures. The resulting algorithms are typically both specific to the implemented operations and not composable.

Transactional memory

Transactional memories (TMs) reduce the number of observable thread interleavings by providing the illusion that all of the reads and writes inside a transaction occur instantaneously. This is implemented using optimistic concurrency control. Transactional reads and writes are tracked. If a transaction attempt is discovered to be non-atomic or if a deadlock is found, the attempt is rolled back and retried. Tracking can be implemented directly in hardware [45, 37, 65, 81], purely in software [44], or in various combinations thereof [78, 55].

TM's promise of simple semantics is complicated by the possibility of rollback and by the limitations of practical implementations. Hardware TMs that can support unbounded transaction size and duration are prohibitively complex. Software TMs that can access all shared mutable state have struggled to simultaneously provide strong semantics and good performance [80, 17]. The most successful uses of TM so far have been in functional languages where most data is immutable [39, 48].

1.3 Picking a Programming Model

In this section we discuss some questions that partition the problem domain into subspaces where different parallel programming models are suitable.

1.3.1 Is the application a deterministic computation?

Shared memory multi-threading adds non-determinism to a program's execution. This makes reasoning, debugging and testing more difficult. If an application domain is already non-deterministic, however, these costs already have been paid. It is therefore natural to first divide applications by whether or not they are deterministic.

There is an important distinction between natural determinism in the application specification and deterministic execution of an implementing program. Pedantically we can say that all executions of a multi-threaded program are deterministic by claiming that interrupts, clock values, arriving network requests, memory bus transactions, instruction interleavings, and cosmic rays are inputs to the program. Practically, we can say that only those inputs

and outputs modelled in the application should be considered. Even modelled inputs may be a source of practical nondeterminism if there is no means to replay them. We will therefore say: *a deterministic application is one that has a single correct output per input, where the input and output are part of the specification and the input can replayed precisely*.

Separating determinism in the specification from determinism in the implementing program has an additional subtlety: a program can be considered correct even if it does not implement the intuitive specification exactly. For example, $\sum_1^n x_i$ is clearly a deterministic specification, but an implementation that uses floating point arithmetic will produce only an approximation of the true sum. Also, the specification's addition is associative but floating point additions are not, so the order in which floating point additions are performed will affect the result. These gaps between high-level specification and implementation might be resolved if the specification is given in terms of the exact sequence of floating point arithmetic operations that should be performed. The insightful reader will notice that this specification is a sequential executable program, and may suggest defining determinism by an equivalence between sequential and parallel executions of the parallel program. Because we are trying to categorize applications rather than implementations, however, we will instead allow a specification to be deterministic even if the implementing programs actually perform a different computation.

1.3.2 How big is the problem?

We can use parallel programming to pool the compute, memory, disk and I/O capacities of a group of machines. This allows us to tackle problems that don't fit on any available shared memory computers, or to build a system from nodes that are less expensive in aggregate than a similarly-sized individual instance.

If parallel programming is used to pool disk and I/O then the model must include those accesses in its model of locality. Because moving persisted data is very expensive, this is usually done by arranging for computations to occur near to the disk or I/O resources that they will access. MPI and the MapReduce model both explicitly model data locality for parallel I/O. Relational databases and high-level DSLs such as Liszt have sufficient information about locality that they can use parallel I/O in their implementations.

1.3.3 Do data accesses have high locality?

Partitioned address spaces require that mutable data be assigned to a particular process or agent. If any other entity needs to access that data it must communicate with the owner, which is cumbersome and expensive if it is frequent. Shared memory makes access equally easy from each thread, at the expense of requiring some sort of concurrency control to prevent conflicts.

1.3.4 Is the problem size limited by available memory?

Models that do not expose destructive writes must fall back on a dynamic garbage collector when static analysis cannot prove that the types are used in a linear fashion. In the extreme case, dynamic garbage collection in a functional programming language can be considered the dual of the alias analysis required for automatic parallelization of imperative code. In practice this results in heavy use of the garbage collector.

Garbage collectors are a classic time-space tradeoff; they have good execution efficiency as long as they are given extra heap space. If a problem's dynamic working set occupies the bulk of the RAM of a parallel system, then programming models that provide in-place destructive updates will be most suitable.

1.3.5 Is the application regular?

We can subdivide deterministic problems by whether their computation space is regular, by which we mean there is a statically known subdivision of the problem space into evenly-sized pieces of work whose memory access patterns are known ahead of time. Regular computation spaces can arise from regular data, as in a dense matrix computation, or from an evenly divided enumeration of a numeric range, as in a grid search optimization. Irregular computation spaces might represent irregular data, recursive traversals, or imbalanced computations over a regular data space.

Irregular applications may require that tasks be moved between parallel contexts to avoid load imbalance. This is more expensive in systems that do not have shared memory. On the other hand regular applications can be statically partitioned, which is often more

efficient.

1.3.6 What is the ratio of reads to writes?

In many cases it is easier to isolate readers from writers than to provide effective scalable concurrency control for concurrent writers. This can take the form of scalable coarse-grained reader-writer locks, snapshot isolation or optimistic concurrency control.

1.3.7 Are communication patterns statically known?

Communication is required for non-local memory accesses and inter-processor synchronization. Process improvements have decreased the power required for computation more quickly than they have decreased the power required for non-local communication. For many applications memory accesses take more power than instruction execution.

Static communication patterns have both usability and execution advantages. At compile time accesses can be analyzed to prove the absence of data races, or to guarantee execution with deterministic outputs. At runtime memory can be prefetched to hide latency, reads and writes can be batched together to improve efficiency, and cache flushes and updates can be performed in software on systems that do not have hardware coherence across all memory accesses. Coherent memory systems have a scalability limit, and are generally assumed to require more power than incoherent ones.

1.3.8 Is a computation defined in terms of phases?

Computations that model time as a sequence of steps or discrete intervals map easily to barriers, and to programming models that contain implicit barriers. Many phased computations can be formulated so that each memory location can have at most one valid value per phase, which can be used to deterministically resolve all memory accesses. Such formulations often allow inter-node communication to be performed exclusively at phase boundaries.

1.3.9 What kind of failure handling is required?

Large scale parallelism dramatically increases the probability of hardware failure, because it dramatically increases the quantity of hardware that can be brought to bear on a single problem. Deterministic parallel computations can be restarted if they encounter a failure, but this strategy becomes less useful as the underlying system becomes larger. At extreme scales the mean time to component failure may be significantly less than the time needed to complete a run.

Non-deterministic applications may also desire tolerance of or recovery from software bugs. Software faults may themselves be non-deterministic (such as a lock cycle leading to a potential deadlock), in which case they may go away if a portion of the program is retried. The system state that led to a problem might not be modelled by the application (such as fragmentation of a garbage collector's heap), in which case migrating or restarting the application might resolve the problem. Behavior in response to failures is a whole-system property; it must be deeply integrated in the design.

1.3.10 Successful uses of restricted models

Let's go over some examples where unrestrained threads are not the best solution.

Sharding in Snort

The Snort application is an intrusion detection system (IDS) for IP networks. It forwards packets unless they are identified to be malicious. Unlike a stateless firewall, an IDS reassembles packet fragments to perform checks of application-level protocols. This allows it to filter HTTP requests, for example. Reassembly is stateful.

Snort's initial implementation is single threaded. For some configurations this is not sufficient to saturate the forwarding machine's networking hardware, so as part of a major rewrite of Snort (3.0) parallelism support was added.

An early parallel version used the producer–consumer model, with stream assembly and filter application being applied in separate phases. The developers reported that this actually reduced total throughput due to an increase in cache misses. The final implementation uses sharding, where traffic is deterministically assigned to a thread and all processing

is performed on that thread. This allows stateful processing of each stream and minimizes inter-thread communication, while allowing threads to share configuration information and a single management interface [76].

Replicated updates in a foreign exchange trading platform

A foreign exchange trading platform accepts bids (requests to buy) and offers (requests to sell) from participants, screens the requests according to credit availability, matches buyers and sellers, and provides streaming updates of the book (the set of outstanding bids and offers). Parallelization of order matching is particularly demanding due to high data contention (most of the activity occurs for just a few instruments) and strong fault-tolerance requirements. As an additional constraint trading rules are often written to imply sequential consistency.

The author is personally familiar with the use of replicated updates for this problem domain. We implemented a system that captures the trading history for a pair of currencies as a list of immutable data structures. Matching for a particular financial instrument is performed by a single thread that constructs and appends a new "state of the world" to this list. Most of the work in the system is performed by tasks that read the contents of the list. In the described system updates are propagated throughout a cluster. This allows the read-only computation processes to be spread across many nodes, and it allows any node in the system to take over the writer's role if there is a failure.

This foreign exchange system also makes extensive use of speculation to hide I/O latency. Updates performed by in-memory transactions are speculatively forwarded to subsequent transactions, as in TLS. Notification to end users is delayed until after operations have been durably written to multiple disks.

Dense matrix multiplication

Dense matrix multiplication forms the core of many practical algorithms. Its specification is deterministic[3]. Its data accesses have good locality if the computation is blocked, and

[3]The order that floating point arithmetic is performed can affect the result, but this is often ignored for domains whose data is well conditioned.

those accesses are completely regular and can be separated into phases. Each parallel context can perform many reads and arithmetic operations for each write. Naive dense matrix multiplication is $O(n^3)$, and the best algorithmic complexity for a practical algorithm (once the constant factor has been included for feasible problem sizes) is $O(n^{2.807})$, so the working set for any dense matrix multiplication that can be tackled today is small enough to comfortably fit in memory.

Dense matrix multiplication can be efficiently solved by most of the parallel programming models covered in Section 1.2. The exceptions are SQL[4], in which the language's inability to iterate over column names is problematic; producer–consumer parallelism with static stages, in which all-to-all communication can only be performed at a stage transition but the number of required communications is dynamically determined by the input; and replicated updates, because all workers need to perform writes. Unstructured thread techniques may be used, but are not necessary. The most efficient solutions may combine several techniques, using vector instructions within the multiplication's inner loop, deterministic phased execution among multiple threads of a processor, and message passing between nodes. These techniques may be exposed directly to the programmer, or they may be targets for a unified model such as Sequoia.

Physical simulations over a static space

Many physical simulations represent a numeric approximation to partial differential equations defined over a continuous space. Solution methods must sample the space and the time domains. When space is sampled regularly in a grid and time is advanced in fixed intervals then the communication patterns are regular and easy to characterize; the resulting computation has a lower arithmetic intensity than dense matrix multiplication, which increases the problem sizes that can be feasibly tackled. This means that the largest simulations scale beyond shared-memory processors. Communication patterns may still be statically determined when space is subdivided into a static mesh, although care must be taken to avoid load imbalance. Large simulations over a static irregular mesh require either a programming model that explicitly models locality and communication (such as MPI,

[4]We consider here only solutions in which tuple elements are primitive values, not vectors or matrices.

X10 or Sequoia) or a system that can analyze the irregular communication pattern. This is a very difficult problem in general; it is made easier by using a domain-specific language in which the mesh is a first-class entity (such as Liszt).

Embarrassing parallelism in Monte Carlo analysis

Monte Carlo analysis computes an approximation of a probability distribution by evaluating a model at randomly chosen points in the input space. This requires sampling many points of the input space. Each evaluation is independent, and because input points are chosen randomly parallel contexts need only to communicate their output. Typically the number of samples is very large, so it can easily fill the parallel machine with uniform work.

Intra-operation parallelism with sequential semantics in MatLab

MatLab's Parallel Computing Toolbox provides parallel execution using threads, MPI or CUDA, while maintaining source-level compatibility with legacy MatLab code. It does this by providing alternate vector and matrix constructors that distribute the underlying data across locations, and then providing parallel execution of large individual operations. (Functions are also provided for using the lower-level constructs directly, but these do not provide a higher-level parallelism model.) This intra-operation parallelism is effective for speeding up programs that spend most of their time in computational kernels. Code optimization in MatLab has historically involved replacing loops and conditionals with bulk vector and matrix operations, which is likely to improve scalability as well. This approach works best for regular applications.

Inter-operation parallelism in DSLs

Automatic parallelization by the compiler of fine-grained sequential code has only proved useful in limited contexts, mainly due to the difficulty of separating the fundamental data dependencies and anti-dependencies from incidental ones. One way to make this problem simpler is to perform the analysis on coarse-grained operations. Bulk matrix and vector operations are a natural start; this approach can be generalized by providing tools that let

library designers provide information to the compiler. We refer to the result as implicitly parallel Domain Specific Languages (DSLs).

Stanford's Pervasive Parallelism Lab is actively researching tools to help library writers build implicitly parallel DSLs. Their DELITE toolkit uses compile-time analysis to build blocks that are then dynamically scheduled [21].

Hadoop for business analytics

Internet advertising produces enormous volumes of data. Each time that an ad is presented there is an opportunity to record many pieces of information that can be used to identify the user and the associated page. This data can be combined with click-through rates to identify successful ad or product strategies, but the quantity of data gathered by a large advertiser necessitates parallel evaluation of business analytics queries.

Analysis of large volumes of read-only data is a good fit for the MapReduce model. The most important performance concern is intelligent parallel I/O; the computation performed for each data element is not expensive. A complete solution for this problem might consist of a cluster of inexpensive Linux servers running Hadoop, with disk and node failures handled in software rather than via expensive hardware.

Parallel chess using Cilk

Exploring potential moves in chess provides ample opportunity for parallelism. The structure is naturally recursive and imbalanced, making it a good fit for fork-join execution with work stealing. StarTech is a parallel chess program written Cilk. In 1993 it tied for third place at the ACM International Computer Chess Championship while running on a 512-processor CM-5. Because competitive chess is played with a time limit, StarTech has a unique scalability measure: chess rating as a function of processor count [56]!

Telecommunications switches in Erlang

The Erlang language was originally developed inside Ericsson for use in soft-real-time environments with extreme reliability requirements. Programs are written as a collection of actors communicating via messages. Each actor is written in a functional single-assignment

style, and the language has built-in support for error recovery and code swapping. In Ericsson's telecommunication switches, Erlang's failure handling is used in a hierarchical style so that the failure of one service does not degrade other operations. The AXD301 switch was reported to provide 9-nines ($1 - 10^{-9}$) uptime, which corresponds to 31 milliseconds of downtime per year.

1.4 Shared Memory Multi-Threading

Shared memory multi-threading imposes the fewest restrictions on the programmer. It exposes the parallel execution contexts directly to the programmer, it allows all threads to read and write all memory, and it allows participants to synchronize or communicate with each other in arbitrary groups at any time.

How do we reconcile the seeming contradiction that we recommend using the most restrictive model possible, while our research focus is on the least restrictive model? There are practical reasons why shared memory multi-threading will continue to be ubiquitous, and hence important:

- Mutable state allows unstructured non-local communication between components, which provides flexibility and agility to software;
- Threads add parallelism to imperative programming languages without changing existing component structuring techniques;
- The recent push for parallelism is driven by multi-core parallel processors, which can implement shared memory more easily than multiprocessors; and
- Threads with shared memory are a good generic substrate for higher level models.

1.4.1 Flexible communication

There are three ways in which components in an imperative sequential program can communicate. They can pass data as arguments to a function[5] call, they can return values from

[5] We use the term function here loosely to also include methods, procedures and impure functions.

a function, or the sender can write data to shared memory and the receiver can read it. The first two mechanisms allow data to be passed only between caller and callee. Non-exceptional multi-hop communication via parameters and return values requires the cooperation of all of the intervening parties, which complicates the code. If the initial design did not envision the need for communication between two components, then this style of communication may require changes to many call sites. Reads and writes to shared state, however, do not appear in the signature of a component, so additional communication pathways can be added more easily. Flexibility reduces the need for up-front design, and it reduces the cost of system iteration.

1.4.2 Evolution of existing languages

Consider the problem of a language designer that wishes to add support for concurrent execution to their imperative sequential language: if parallel contexts are not able to communicate via the existing well known and well used pathway, the entire character of the language and its libraries will be changed. Multi-threaded access to shared memory has the favorable property that it can be made to work within existing paradigms.

When threads were originally introduced they were primarily a mechanism for tolerating I/O latency on systems that actually could only execute one thread at a time. In this context they are easy to use effectively; coarse-grained locking has low overheads and does not limit parallelism when there is none. Once multiple execution contexts became available in hardware, however, scalability became a concern. This thesis tackles the complexity that arises when implementing scalable programs.

1.4.3 Intra-chip coherence

The resurgence of parallel programming is being driven by the switch to multi-core processors. Multi-core processors have essentially replaced sequential processors in the desktop and server space, and are common even in embedded applications. Intra-chip communication has higher bandwidth, lower latency and lower power requirements than inter-chip communication, so the costs of maintaining a single coherent address space are lower than

they would be for a multiprocessor with the same number of threads. In addition, by keeping cores coherent with each other they can share a last-level cache, maximizing cache utilization and providing efficient inter-core communication.

1.4.4 A generic substrate

One compelling reason to study shared memory multi-threading is that it can serve as a greatest common denominator for other parallel programming models. Threads communicating through shared memory can be used to implement all of the models from Section 1.2. Databases, P-Linq, Haskell, Unix pipes[6], GRAMPS, BLAS, DELITE, Sequoia, Phoenix, MPI, Erlang, and Scala Actors all have at least one implementation that uses threads and that communicates (inside the implementation) using shared memory.

Scala's actors are a particularly interesting case. They are implemented purely as a library inside a language that allows unrestricted threads and shared mutable state. While this means that there is no compile-time checking that actors are not also using other forms of communication, Scala's powerful language features make the syntax feel native. Programs that use Scala actors can also harness parallel execution using lazy functional evaluation, fork-join parallelism using the JSR166Y framework [58], implicit intra-operation parallelism for operations on bulk data [71] and arbitrary threading using Java's concurrent collection classes or protected by locks or STM [16].

While a multi-paradigm approach such as Scala's allows each component of an application to choose the parallelization model that fits best, it prevents or complicates optimization techniques that rely on a specific parallelism structure. For example, the Erlang language provides no way to construct a pointer from one thread's data to another, so garbage collection can be performed separately for each thread's heap. Scala's actors all operate inside a single heap, making incremental garbage collection much more difficult. Polymorphic staging is an active research area that addresses this challenge by allowing type-safe libraries to participate in code generation [77].

[6]While Unix pipes use processes and message passing at the user level, inside the kernel they use hardware threads and shared memory buffers.

1.4.5 Platform and language choice

What threading platform and language should we use as a foundation for our research? One of the most important practical distinctions in imperative programming languages is whether they have automatic garbage collection. The idiomatic style used in languages with manual memory allocation largely uses patterns that allow object lifetimes to be computed using local information. While restrictive, one useful side effect of this programming style is that deterministic lifetime information can be used for resource management. Garbage collected languages ease sharing of immutable data and free the user from the reasoning required for manual deallocation, but don't provide a bound on the lifetime of unreachable resources.

POSIX threads

Virtually all desktop computers and servers start with an abstraction of hardware processing contexts and a single coherent memory space. The operating system then virtualizes these primitives by subdividing memory (typically using hardware features to limit inter-program memory accesses) and multiplexing software threads among the available processors. The POSIX threads library (also known as pthreads) is a minimal C API for this abstraction [60]. It provides the ability to create new threads, and it integrates with the operating system's scheduler so that threads that are waiting for an event do not need to occupy a physical execution context.

POSIX threads are supported either natively or through an efficient translation layer on many operating systems, including Windows, OpenVMS, Linux, and the Unix dialects FreeBSD, NetBSD, Mac OS X, AIX, HP/UX and Solaris. As a result, programs written using pthreads are highly portable.

JVM and CLR threads

Oracle's JVM and Microsoft's CLR are sophisticated virtual machines that perform just-in-time compilation of bytecodes. Both the JVM and the CLR are targeted by many languages, and their threading models are similar. The high level conclusions of parallelism research using one VM should apply directly to the other.

JVM threads have the same semantics as POSIX threads, and support the same basic operations. In the HotSpot JVM each `java.lang.Thread` is actually executed on exactly one `pthread_t`. The monitors provided by Java bytecode are just ordinary reentrant mutexes with the additional constraint that entry and exit must be statically paired.

Although the JVM's natively supported threads and coordination primitives closely correspond to those in pthreads, it provides two powerful features that are not yet widely available in C or C++: a well-defined memory model and automatic garbage collection. Java's memory model defines the allowed behaviors when multiple threads access the same memory location, striking a balance between simple semantics and optimization opportunities [50]. Garbage collection is essential for optimistic concurrency and lock-free algorithms. If it is not provided natively then it must be implemented manually. Even though an algorithm-specific solution may be simpler than a generic collector this is still a substantial engineering effort. Hazard pointers for lock-free data structures [64] and delayed memory deallocation in software transactional memory [24] are both algorithm-specific collectors.

Garbage collection is also useful for implementing language features such as closures and continuations that make library-based parallel programming models appear to use native syntax. For example CCSTM uses features of the Scala programming language to make it appear that `atomic` is a keyword, but it is actually implemented as a function whose argument is an anonymous function that uses a block syntax.

Language

Statically typed languages are more likely to achieve peak performance from the JVM or CLR, because types are an integral part of those VMs' bytecodes and optimization strategies. We have chosen to use a combination of Java and Scala in our research. Java provides maximum performance on the JVM, as its statements map almost directly onto JVM bytecodes. Scala uses a combination of type inference and syntactic sugar to improve on Java's verbosity while still generating efficient bytecode for the careful programmer.

While our techniques could be adapted to unmanaged languages, our work focuses almost exclusively on managed languages. We have no reason to expect that our results would be different if the host language was C# or F#, or if the Scala code had been written in (more verbose) Java.

1.5 Our Vision for Multi-Threaded Programming

Our vision of multi-threaded programming consists of five parts: immutable data structures that can be used as values; efficient snapshots of mutable data structures; composable atomicity for operations on shared mutable state; type system involvement; and strong-by-default semantics. We are inspired by Tim Sweeney's layering of purely functional data structures and STM [87].

1.5.1 Immutable data structures

Good support for efficient immutable data structures allows much of the complexity of a program to be located in functions that map an unchanging input to an unchanging output. The most obvious concurrency benefit to performing operations on unchanging inputs is that thread scheduling doesn't affect the result. This isolation simplifies the implementation, at least for the many cases where the output is still useful when it is completed. Unchanging data also provides the advantage of referential integrity, which allows the programmer to assume that the same name (in the same scope) always refers to the same value. Without referential integrity, the intervening code may alter data, reducing opportunities for localized correctness reasoning.

One way to guarantee referential integrity is to use immutable (or persistent) data structures, in which every operation constructs a new value without destroying the old. Although the use of a single-assignment language makes it easier to code in this style (partly by making it harder to do anything else), it is also applicable to imperative languages that are founded on mutation. Java's `String` class is perhaps the most widely used immutable data structure. Internally it is usually implemented using a mutable **`char`** array, but it makes use of encapsulation to provide value semantics.

1.5.2 Snapshots for mutable data structures

Snapshots of mutable data structures provide most of the advantages of isolation while still supporting imperative programming's unstructured communication patterns. This allows the benefits of isolation to be introduced to existing code and existing languages without

requiring a conversion to purely functional programming.

Snapshots often allow a more efficient implementation than a fully persistent data structure, because the explicit snapshot operation gives the system information about which versions of the data structure must be preserved. In a system using lazy copy-on-write, for example, objects that are unshared can be efficiently identified and updated in place. This allows many updates to be performed without memory allocation or deallocation. Snapshots are never less efficient than an immutable data structure, because they can be trivially implemented as a mutable reference to the immutable implementation.

1.5.3 Composable operations

Snapshot isolation is not sufficient for structuring all concurrent operations, because the result of an isolated computation may be out-of-date before it is completed. This *write skew* is especially problematic for operations that perform updates to shared data.

Composability is in general desirable; the question is whether the advantages of a particular approach outweigh the disadvantages. In this thesis we will demonstrate that for an important class of data structures (sets and maps) the performance overheads are low and any additional complexity can be centralized in a shared library. Importantly, we also will demonstrate that composable operations can be introduced incrementally, with the resulting data structures providing competitive performance when implementing the interface of their non-composable counterpart.

1.5.4 Help from the type system

Java's **`final`** and C/C++'s **`const`** are already widely used to simplify reasoning about concurrent mutation. If a field is marked **`final`** then the program may assume that no concurrent modification is possible. **`const`** does not directly guarantee that another part of the program can't modify a memory location, because a non-**`const`** reference may also exist or may be created with a typecast, but it provides the same practical benefits when combined with a programming methodology that limits explicit casts and avoids sharing of mutable pointers and references.

This thesis is not a user study on the productivity impact of enforcing concurrency

safety using the type system, nor a religious document that proscribes the "correct" amount of compile-time versus run-time checking. The authors' personal experience, however, has made them wary of attempts to provide automatic concurrency control for every mutable memory location in a program.

We have previously studied the use of run-time profiling and dynamic code rewriting to reduce the overhead of software transactional memory [17]. Our result was mostly successful, but its opaque performance model and high complexity convinced us that it was better to mark transactional data explicitly. Explicit annotation of data whose mutation must be composable allows strong semantics to be reliably provided with no performance overhead for non-transactional data.

Leveraging the type system does not necessarily require changes to the language. The STM provided by Haskell uses mutable boxes and a monad to statically separate transactionally-modifiable data with normal Haskell constructs. CCSTM and ScalaSTM use a similar strategy for Scala, albeit with fewer guarantees proved by the compiler.

1.5.5 Strong-by-default semantics

In general, strong correctness models such as linearizability or strong atomicity require more communication that relaxed consistency models. This can make strong semantics less performant or less scalable than weak ones[7]. Nevertheless, it is our view that strong semantics should be the default.

Our preference for strong semantics is rooted in our experience in testing and debugging software systems. We have observed that when an interface allows two behaviors whose frequency differs dramatically, the infrequent behavior is more likely to be a source of bugs. In sequential code we have learned to look carefully at error handling code and at conditions such as buffer overflows, hash collisions, or memory allocation failures. While extra care is required, these rare conditions are repeatable and hence amenable to testing.

Rare interleavings in a concurrent system, however, are notoriously difficult to enumerate, reproduce, and test. The author's initial interest in STM arose after he spent three

[7] In a distributed setting strong correctness guarantees must also be traded against availability, but currently multi-threaded systems are usually considered to be a single failure domain.

weeks over the course of a year to debug a single race condition in a lock-based concurrent system; the bug never manifested in a development or Q&A environment but appeared about once a month in production servers.

Our view is that the default behavior for an operation's access to shared memory is that it should appear to be atomic, and that this atomicity should hold independently of the accesses performed by other code. Only after profiling has demonstrated that these strong semantics actually lead to an observable reduction of performance or scalability should localized correctness criteria be abandoned. In this thesis we demonstrate that strong semantics and efficiency are not mutually exclusive.

1.6 Compound Atomic Operations

Concurrent collection classes have emerged as one of the core abstractions of parallel programming using threads. They provide the programmer with the simple mental model that most method calls are linearizable, while admitting efficient and scalable implementations. This efficiency and scalability, however, comes from the use of non-composable concurrency control schemes that are tailored to the specific data structure and the operations provided. Full composability can be provided by utilizing a software transactional memory (STM), but the performance overheads make this unattractive.

In this thesis we improve on the limited primitives provided by existing concurrent sets and maps both by adding new primitives and by providing full automatic composability. In Chapter 2 we add a powerful new operation (`clone`) to the operations previously supported by efficient thread-safe maps. In Chapter 3 we implement efficient thread-safe maps whose operations can be composed with STM. In Chaper 4 we introduce and evaluate a data structure that provides both `clone` and full STM integration, while retaining good performance. Chapter 4 introduces the idea of combining lock-free algorithms with TM.

1.7 Our Contributions

This thesis presents new algorithms and techniques that make thread-safe sets and maps easier to use, while retaining excellent performance and scalability. At a high level we

contribute:

- SnapTree – the first thread-safe ordered map data structure that provides both lazy snapshots and invisible readers. The snapshots allow consistent iteration and multiple atomic reads without blocking subsequent writes. Invisible readers are important for scalability on modern cache-coherent architectures (Chapter 2) [14].

- Transactional Predication – software transactional memory (STM) integration for existing non-composable thread-safe maps, without the performance penalty usually imposed by STM. Transactionally predicated collections used outside a software transaction have performance competitive with mature non-composable implementations, and better performance than existing transactional solutions when accessed inside an atomic block (Chapter 3) [15].

- Predicated SnapTrie – a hash trie that provides both lazy snapshots and full transactional memory integration. This data structure provides both the efficient atomic bulk reads of lazy copy-on-write and the efficient atomic compound updates of transactionally predicated concurrent maps. In developing this data structure we give conditions under which a lock-free algorithm can be simultaneously used inside and outside transactions, and we show how the lock-free algorithm can be made simpler by falling back to TM for exceptional cases (Chapter 4).

Chapter 2

SnapTree

2.1 Introduction

The widespread adoption of multi-core processors places an increased focus on data structures that provide efficient and scalable multi-threaded access. These data structures are a fundamental building block of many parallel programs; even small improvements in these underlying algorithms can provide a large performance impact. One widely used data structure is an ordered map, which adds ordered iteration and range queries to the key-value association of a map. In-memory ordered maps are usually implemented as either skip lists [73] or self-balancing binary search trees.

Research on concurrent ordered maps for multi-threaded programming has focused on skip lists, or on leveraging software transactional memory (STM) to manage concurrent access to trees [44, 4]. Concurrent trees using STM are easy to implement and scale well, but STM introduces substantial baseline overheads and performance under high contention is still an active research topic [2]. Concurrent skip lists are more complex, but have dependable performance under many conditions [30].

In this chapter we present a concurrent relaxed balance AVL tree. We use optimistic concurrency control, but carefully manage the tree in such a way that all atomic regions have fixed read and write sets that are known ahead of time. This allows us to reduce practical overheads by embedding the concurrency control directly. It also allows us to take

advantage of algorithm-specific knowledge to avoid deadlock and minimize optimistic retries. To perform tree operations using only fixed sized atomic regions we use the following mechanisms: search operations overlap atomic blocks as in the hand-over-hand locking technique [7]; mutations perform rebalancing separately; and deletions occasionally leave a routing node in the tree. We also present a variation of our concurrent tree that uses lazy copy-on-write to provide a linearizable `clone` operation, which can be used for strongly consistent iteration.

Our specific contributions:

- We describe hand-over-hand optimistic validation, a concurrency control mechanism for searching and navigating a binary search tree. This mechanism minimizes spurious retries when concurrent structural changes cannot affect the correctness of the search or navigation result (Section 2.3.3).
- We describe partially external trees, a simple scheme that simplifies deletions by leaving a routing node in the tree when deleting a node that has two children, then opportunistically unlinking routing nodes during rebalancing. As in external trees, which store values only in leaf nodes, deletions can be performed locally while holding a fixed number of locks. Partially external trees, however, require far fewer routing nodes than an external tree for most sequences of insertions and deletions (Section 2.3.5).
- We describe a concurrent partially external relaxed balance AVL tree algorithm that uses hand-over-hand optimistic validation, and that performs all updates in fixed size critical regions (Section 2.3).
- We add copy-on-write to our optimistic tree algorithm to provide support for an atomic `clone` operation and snapshot isolation during iteration (Section 2.3.10).
- We show that our optimistic tree outperforms a highly-tuned concurrent skip list across many thread counts, contention levels, and operation mixes, and that our algorithm is much faster than a concurrent tree implemented using an STM. Our algorithm's throughput ranges from 13% worse to 98% better than the skip list's on a variety of simulated

read and write workloads, with an average multi-threaded performance improvement of 32%. We also find that support for fast cloning and consistent iteration adds an average overhead of only 9% to our algorithm (Section 2.5).

2.2 Background

An AVL tree [1] is a self-balancing binary search tree in which the heights of the left and right child branches of a node differ by no more than one. If an insertion to or deletion from the tree causes this balance condition to be violated then one or more rotations are performed to restore the AVL invariant. In the classic presentation, nodes store only the difference between the left and right heights, which reduces storage and update costs. Balancing can also be performed if each node stores its own height.

The process of restoring the tree invariant becomes a bottleneck for concurrent tree implementations, because mutating operations must acquire not only locks to guarantee the atomicity of their change, but locks to guarantee that no other mutation affects the balance condition of any nodes that will be rotated before proper balance is restored. This difficulty led to the idea of relaxed balance trees, in which the balancing condition is violated by mutating operations and then eventually restored by separate rebalancing operations [35, 69, 57]. These rebalancing operations involve only local changes. Bougé et al. proved that any sequence of localized application of the AVL balancing rules will eventually produce a strict AVL tree, even if the local decisions are made with stale height information [11].

Binary search trees can be broadly classified as either internal or external. Internal trees store a key-value association at every node, while external trees only store values in leaf nodes. The non-leaf nodes in an external tree are referred to as *routing nodes*, each of which has two children. Internal trees have no routing nodes, while an external tree containing n values requires n leaves and $n-1$ routing nodes.

Deleting a node from an internal tree is more complicated than inserting a node, because if a node has two children a replacement must be found to take its place in the tree. This replacement is the successor node, which may be many links away (y is x's successor in Figure 2.1). This complication is particularly troublesome for concurrent trees, because this means that the critical section of a deletion may encompass an arbitrary number of

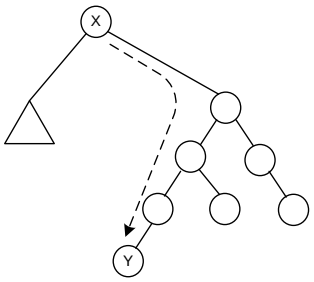


Figure 2.1: Finding the successor for deletion of an internal node.

nodes. The original delayed rebalancing tree side-stepped this problem entirely, supporting only `insert` and `search` [1]. Subsequent research on delayed rebalancing algorithms considered only external trees. In an external tree, a leaf node may always be deleted by changing the link from its grandparent to point to its sibling, thus splicing out its parent routing node (see Figure 2.8.b).

While concurrent relaxed balance tree implementations based on fine-grained read-write locks achieve good scalability for disk based trees, they are not a good choice for a purely in-memory concurrent tree. Acquiring read access to a lock requires a store to a globally visible memory location, which requires exclusive access to the underlying cache line. Scalable locks must therefore be striped across multiple cache lines to avoid contention in the coherence fabric [62], making it prohibitively expensive to store a separate lock per node.

Optimistic concurrency control (OCC) schemes using version numbers are attractive because they naturally allow invisible readers, which avoid the coherence contention inherent in read-write locks. Invisible readers do not record their existence in any globally visible data structure, rather they read version numbers updated by writers to detect concurrent mutation. Readers 'optimistically' assume that no mutation will occur during a critical region, and then retry if that assumption fails. Despite the potential for wasted work, OCC can provide for better performance and scaling than pessimistic concurrency control [79].

Software transactional memory (STM) provides a generic implementation of optimistic concurrency control, which may be used to implement concurrent trees [44] or concurrent relaxed balance trees [4]. STM aims to deliver the valuable combination of simple parallel programming and acceptable performance, but internal simplicity is not the most important

goal of a data structure library.[1] For a widely used component it is justified to expend a larger engineering effort to achieve the best possible performance, because the benefits will be multiplied by a large number of users.

STMs perform conflict detection by tracking all of a transaction's accesses to shared data. This structural validation can reject transaction executions that are semantically correct. Herlihy et al. used early release to reduce the impact of this problem [44]. Early release allows the STM user to manually remove entries from a transaction's read set. When searching in a binary tree, early release can mimic the effect of hand-over-hand locking for successful transactions. Failed transactions, however, require that the entire tree search be repeated. Elastic transactions require rollback in fewer situations than early release and do not require that the programmer explicitly enumerate entries in the read set, but rollback still requires that the entire transaction be reexecuted [29].

Skip lists are probabilistic data structures that on average provide the same time bounds as a balanced tree, and have good practical performance [73]. They are composed of multiple levels of linked lists, where the nodes of a level are composed of a random subset of the nodes from the next lower level. Higher lists are used as hints to speed up searching, but membership in the skip list is determined only by the bottom-most linked list. This means that a concurrent linked list algorithm may be augmented by lazy updates of the higher lists to produce a concurrent ordered data structure [72, 30]. Skip lists do not support structural sharing, so copy-on-write cannot be used to implement fast cloning or consistent iteration. They can form the foundation of an efficient concurrent priority queue [85].

2.3 Our Algorithm

We present our concurrent tree algorithm as a map object that supports five methods: `get`, `put`, `remove`, `firstNode`, and `succNode`. For space reasons we omit practical details such as user-specified `Comparator`s and handling of `null` values. The `get`(k) operation returns either v, where v is the value currently associated with k, or `null`; `put`(k, v) associates k and

[1] Here we are considering STM as an internal technique for implementing a concurrent tree with a non-transactional interface, not as a programming model that provides atomicity across multiple operations on the tree.

```
1  class Node<K,V> {
2      volatile int height;
3      volatile long version;
4      final K key;
5      volatile V value;
6      volatile Node<K,V> parent;
7      volatile Node<K,V> left;
8      volatile Node<K,V> right;
9      ...
10 }
```

Figure 2.2: The fields for a node with key type *K* and value type *V*.

a non-null v and returns either v_0, where v_0 is the previous value associated with k, or `null`; `remove`(k) removes any association between k and a value v_0, and returns either v_0 or `null`; `firstNode()` returns a reference to the node with the minimal key; and `succNode`(n) returns a reference to the node with the smallest key larger than n`.key`. The `firstNode` and `succNode` operations can be used trivially to build an ordered iterator. In Section 2.4 we will show that `get`, `put`, and `remove` are linearizable [47]. We will discuss optimistic hand-over-hand locking in the context of `get` (Section 2.3.3) and partially external trees in the context of `remove` (Section 2.3.5).

Our algorithm is based on an AVL tree, rather than the more popular red-black tree [6], because relaxed balance AVL trees are less complex than relaxed balance red-black trees. The AVL balance condition is more strict, resulting in more rebalancing work but smaller average path lengths. Pfaff [70] characterizes the workloads for which one tree performs better than the other, finding no clear winner. Our contributions of hand-over-hand optimistic validation and local deletions using partially external trees are also applicable to relaxed balance red-black trees. Lookup, insertion, update, and removal are the same for both varieties of tree. Only the post-mutation rebalancing (Section 2.3.6) is affected by the choice.

2.3.1 The data structure: `Node`

The nodes that compose our tree have a couple of variations from those of a normal AVL tree: nodes store their own height rather than the difference of the heights of the children; nodes for a removed association may remain in the tree with a `value` of `null`; and nodes

```
11 static long Unlinked = 0x1L;
12 static long Growing = 0x2L;
13 static long GrowCountIncr = 1L << 3;
14 static long GrowCountMask = 0xffL << 3;
15 static long Shrinking = 0x4L;
16 static long ShrinkCountIncr = 1L << 11;
17 static long IgnoreGrow = ~(Growing | GrowCountMask);
```

Figure 2.3: Version bit manipulation constants.

contain a version number used for optimistic concurrency control. Figure 2.2 shows the fields of a node. All fields except for `key` are mutable. `height`, `version`, `value`, `left`, and `right` may only be changed while their enclosing node is locked. `parent` may only be changed while the parent node is locked (both the old and the new parent must be locked). The delayed balancing of our tree is a property of the algorithm, and is not visible in the type signature.

For convenience, the map stores a pointer to a *root holder* instead of the root node itself. The root holder is a `Node` with no key or value, whose right child is the root. The root holder allows the implementation to be substantially simplified because it never undergoes rebalancing, never triggers optimistic retries (its `version` is always zero), and allows all mutable nodes to have a non-null parent. The map consists entirely of the `rootHolder` reference.

2.3.2 Version numbers

The version numbers used in our algorithm are similar to those in McRT-STM, in which a reserved 'changing' bit indicates that a write is in progress and the remainder of the bits form a counter [79]. (Our algorithm separates the locks that guard node update from the version numbers, so the changing bit is not overloaded to be a mutex as in many STMs.) To perform a read at time t_1 and verify that the read is still valid at t_2: at t_1 read the associated version number v_1, blocking until the change bit is not set; read the protected value x; then at t_2 reread the version number v_2. If $v_1 = v_2$ then x was still valid at t_2.

Our algorithm benefits from being able to differentiate between the structural change to a node that occurs when it is moved down the tree (shrunk) and up the tree (grown). Some

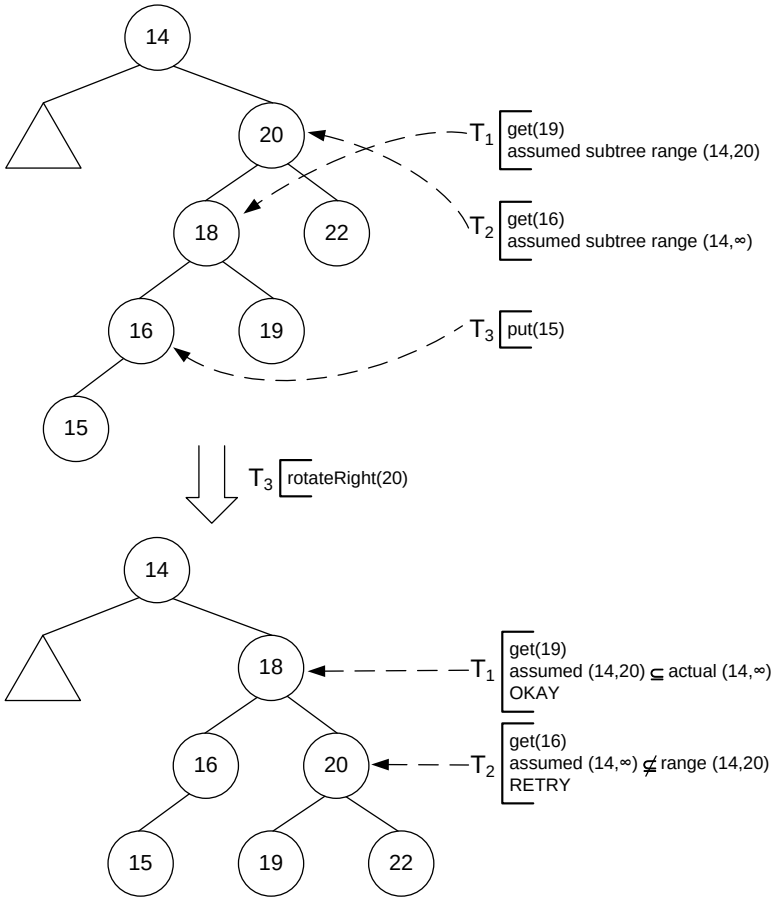


Figure 2.4: Two searching threads whose current pointer is involved in a concurrent rotation. The node 18 grew, so T_1's search is not invalidated. The node 20 shrank, so T_2 must backtrack.

operations are invalidated by either shrinks or grows, while others are only invalidated by shrinks. We use a single 64-bit value to encode all of the version information, as well as to record if a node has been unlinked from the tree. There is little harm in occasionally misclassifying a grow as a shrink, because no operation will incorrectly fail to invalidate as a result. We therefore overlap the shrink counter and the grow counter. We use the most significant 53 bits to count shrinks, and the most significant 61 bits to count grows. This layout causes a grow to be misclassified as a shrink once every 256 changes, but it never causes a shrink to be misclassified as a grow. The bottom three bits are used to implement an unlink bit and two change bits, one for growing and one for shrinking (Figure 2.3).

```
18 static Object Retry = new Object();
19
20 V get(K k) {
21   return (V)attemptGet(k, rootHolder, 1, 0);
22 }
23
24 Object attemptGet(
25       K k, Node node, int dir, long nodeV) {
26   while (true) {
27     Node child = node.child(dir);
28     if (((node.version^nodeV) & IgnoreGrow) != 0)
29       return Retry;
30     if (child == null)
31       return null;
32     int nextD = k.compareTo(child.key);
33     if (nextD == 0)
34       return child.value;
35     long chV = child.version;
36     if ((chV & Shrinking) != 0) {
37       waitUntilNotChanging(child);
38     } else if (chV != Unlinked &&
39                child == node.child(dir)) {
40       if (((node.version^nodeV) & IgnoreGrow) != 0)
41         return Retry;
42       Object p = attemptGet(k, child, nextD, chV);
43       if (p != Retry)
44         return p;
45 } } }
```

Figure 2.5: Finding the value associated with a k.

2.3.3 Hand-over-hand optimistic validation: `get`(k)

If k is present in the map then `get`(k) must navigate from the root holder to the node that holds k. If k is not present in the tree then `get` must navigate to the node that would be k's parent if it were inserted. If no concurrency control is performed, a search may be led astray by a concurrent rotation. The well-known lock-based technique for handling this is hand-over-hand locking (also known as spider locks, lock chaining, chain locking, . . .), which decreases the duration over which locks are held by releasing locks on nodes whose rotation can no longer affect the correctness of the search [7]. With both exclusive locks or read-write locks the root lock must be acquired by each accessor, making it a point of contention. We use optimistic validation to guard critical regions, chaining in a manner similar to hand-over-hand locking. This avoids the scalability problems of the root lock

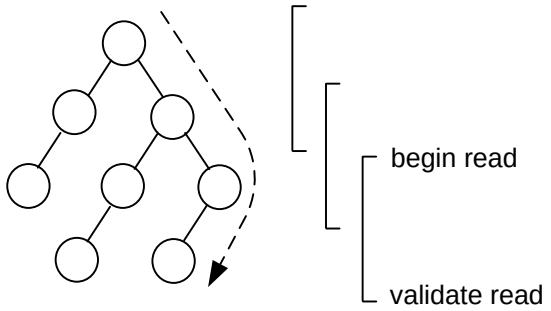


Figure 2.6: Hand-over-hand optimistic validation.

while using only fixed size critical regions (see Figure 2.4).

The key to hand-over-hand optimistic validation is to reason explicitly about the implicit state of a search, which consists of an open interval of keys that must either be absent from the entire tree or present in the current subtree. Each time that the search process performs a comparison and navigates downward, the interval is reduced. At all times the interval includes the target key, so if the subtree ever becomes empty we can conclude that no node with that key is present. The optimistic validation scheme only needs to invalidate searches whose state is no longer valid.

If a search that was valid when it was at node n has since traversed to a child node c of n, and the pointer from n to c has not been modified, then the search is still valid, because the subtree has not been changed. To prevent false invalidations we use separate version numbers to track changes to n`.left` and n`.right`. We reduce storage overheads by actually storing the version number that protects the link from n to c in c`.version`. (This requires us to traverse the link twice, once to locate the version number and once as a read that is actually protected by OCC.) To successfully navigate through a node, there must be a point where both the inbound and outbound link are valid. This is accomplished by validating the version number that protects the inbound link after both the outbound link and its version number have been read.

A search may still be valid despite a change to a child link, if every node in the tree within the computed bounds must still be contained in the subtree. Consider the scenario in Figure 2.4, in which a mutating thread T_3 performs a rotation while two searches are in progress. (We defer a discussion of the visible intermediate states during rotation to

Section 2.3.7.) T_1's search points to the node that is raised by the rotation, which means that its implicit state is not invalidated. Intuitively if a rotation 'grows' the branch rooted at a node, then a search currently examining the node will not fail to find its target. Conversely, if a rotation 'shrinks' the branch under a node, then a search pointing to that node may falsely conclude that a node is not present in the tree. In the latter case the implicitly computed subtree bounds are incorrect. T_2's search points to a shrinking node, which means that it must backtrack to node 14, the previous level of the search. Changes to a child pointer may also be 'neutral' if they preserve the range of the keys contained in the subtree, as can occur during deletion.

Figure 2.5 shows the code for the `get` operation. The bulk of the work is accomplished by `attemptGet`. `attemptGet` assumes that the pointer to `node` was read under version number `nodeV`, and is responsible for revalidating the read at Line 40 after performing a validated read of the child pointer at Line 39. If there is no child then the final validation of the traversal to `node` occurs on Line 28. Note that Line 27's access to the child is not protected, it is merely used to gain access to the version number that protects the inbound pointer. The hand-off of atomically executed regions occurs between the child read and the final validation (Line 28).

`attemptGet` returns the special value `Retry` on optimistic failure, which triggers a retry in the outer call. `get` emulates the first half of a hand-off by pretending to have followed a pointer to `rootHolder` under version 0. `get` does not need a retry loop because the outer-most invocation cannot fail optimistic validation, as `rootHolder.version` is always 0. `IgnoreGrow` is used to ignore changes that grow the subtree, since they cannot affect the correctness of the search. We discuss the `waitUntilNotChanging` method in Section 2.3.9.

Conceptually, the execution interval between reading the version number at Line 35 and the validation that occurs in the recursive invocation at Line 40 constitute a read-only transaction. Each level of the `attemptGet` recursion begins a new transaction and commits the transaction begun by its caller. Interestingly, we can only determine retroactively whether any particular validation at Line 40 was the final validation (and hence the commit) of the atomic region. If a recursive call returns `Retry` the enclosing loop will attempt another validation, which has the effect of extending the duration of the transaction begun

```
46 V put(K k, V v) {
47   return (V)attemptPut(k, v, rootHolder, 1, 0);
48 }
49
50 Object attemptPut(
51       K k, V v, Node node, int dir, long nodeV) {
52   Object p = Retry;
53   do {
54     Node child = node.child(dir)
55     if (((node.version^nodeV) & IgnoreGrow) != 0)
56       return Retry;
57     if (child == null) {
58       p = attemptInsert(k, v, node, dir, nodeV);
59     } else {
60       int nextD = k.compareTo(child.key);
61       if (nextD == 0) {
62         p = attemptUpdate(child, v);
63       } else {
64         long chV = child.version;
65         if ((chV & Shrinking) != 0) {
66           waitUntilNotChanging(child);
67         } else if (chV != Unlinked &&
68                    child == node.child(dir)) {
69           if (((node.version^nodeV) & IgnoreGrow)!=0)
70             return Retry;
71           p = attemptPut(k, v, child, nextDir, chV);
72     } } }
73   } while (p == Retry);
74   return p;
75 }
76 Object attemptInsert(
77       K k, V v, Node node, int dir, long nodeV) {
78   synchronized (node) {
79     if (((node.version^nodeV) & IgnoreGrow) != 0 ||
80         node.child(dir) != null)
81       return Retry;
82     node.setChild(dir, new Node(
83         1, k, v, node, 0, null, null));
84   }
85   fixHeightAndRebalance(node);
86   return null;
87 }
88 Object attemptUpdate(Node node, V v) {
89   synchronized (node) {
90     if (node.version == Unlinked) return Retry;
91     Object prev = node.value;
92     node.value = v;
93     return prev;
94   }
95 }
```

Figure 2.7: Inserting or updating the value associated with k.

by the caller. This 'resurrection' of a transaction that was previously considered committed is only possible because these atomic regions perform no writes.

2.3.4 Insertion and update: `put`(k, v)

The `put` operation may result in either an insertion or an update, depending on whether or not an association for the key already is present in the tree. It starts in exactly the same manner as `get`, because its first task is to identify the location in the tree where the change should occur. We do not extract this common functionality into a separate function, because during insertion we must perform the last check of the hand-over-hand optimistic validation after a lock on the parent has been acquired. Figure 2.7 shows the implementation of `put`.

If we discover that `node.key` matches the target key then we can be trivially certain that any concurrent tree rotations will not affect the ability of the search to find a matching node. This means that on Line 90 of the `attemptUpdate` helper function we do not need to examine `node.version` for evidence of shrinks, rather we only verify that the node has not been unlinked from the tree. The lock on a node must be acquired before it is unlinked, so for the duration of `attemptUpdate`'s critical region we can be certain that the node is still linked.

To safely insert a node into the tree we must acquire a lock on the future parent of the new leaf, and we must also guarantee that no other inserting thread may decide to perform an insertion of the same key into a different parent. It is not sufficient to merely check that the expected child link is still null after acquiring the parent lock. Consider the tree $b(a, d(\cdot, e))$, in which $p(l, r)$ indicates that l is the left child of p and r is the right child of p, $\cdot$ represents a null child link, and the keys have the same ordering as the name of their node. A put of c by a thread T_1 will conclude that the appropriate parent is d. If some other thread performs a left-rotation at b then the tree will be $d(b(a, \cdot), e)$, which may cause a second thread T_2 to conclude that the insertion should be performed under b. If T_2 proceeds, and then later performs a double rotation (left at b, right at d) the resulting tree will be $c(b(a, \cdot), d(\cdot, e))$. To guard against sequences such as this, Lines 79 and 80 of `attemptInsert` perform the same closing validation as in `get`. Any rotation that could change the parent into which k should be inserted will invalidate the implicit range of the

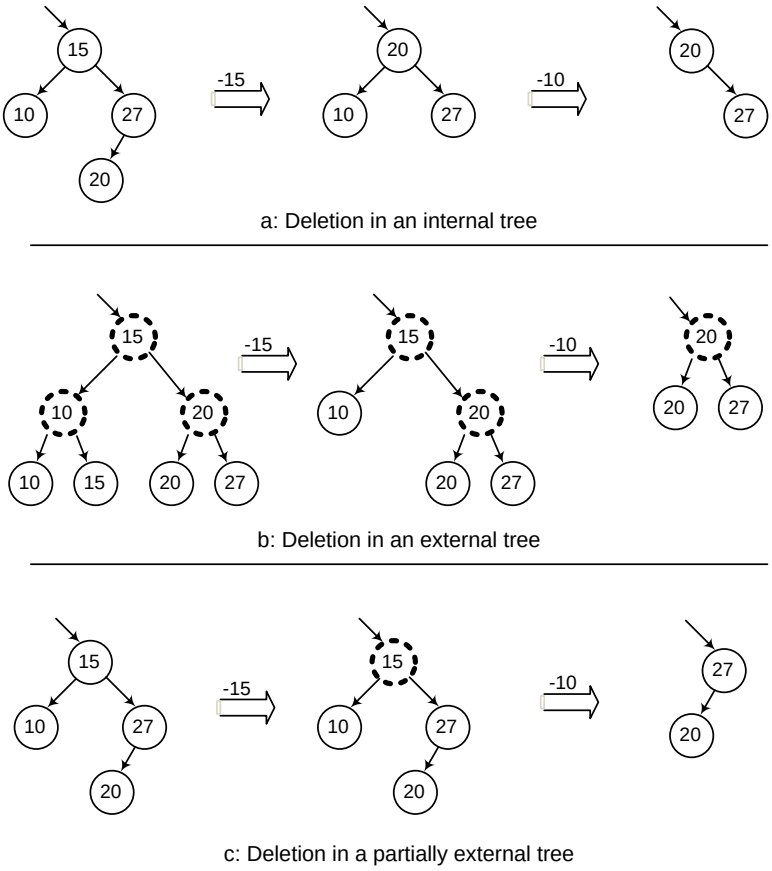


Figure 2.8: A sequence of two deletions in different types of trees.

traversal that arrived at the parent, and hence will be detected by the final validation.

2.3.5 Partially external trees: `remove`(k)

Up until now we have only considered operations that require locking a single node; removals are more difficult. In an internal tree with no routing nodes, deletion of a node n with two children requires that n's successor s be unlinked from n`.right` and linked into n's place in the tree. Locating s may require the traversal of as many as n`.height` -1 links. In a concurrent tree the unlink and relink of s must be performed atomically, and any concurrent searches for s must be invalidated. Every node along the path from n to s's original location must be considered to have shrunk, and hence must be locked. This excessive locking negatively impacts both performance and scalability.

```
96  V remove(K k) {
97    return (V) attemptRemove(k, rootHolder, 1, 0);
98  }
99  ... // attemptRemove is similar to attemptPut
100 boolean canUnlink(Node n) {
101   return n.left == null || n.right == null;
102 }
103 Object attemptRmNode(Node par, Node n) {
104   if (n.value == null) return null;
105   Object prev;
106   if (!canUnlink(n)) {
107     synchronized (n) {
108       if (n.version == Unlinked || canUnlink(n))
109         return Retry;
110       prev = n.value;
111       n.value = null;
112     }
113   } else {
114     synchronized (par) {
115       if (par.version == Unlinked || n.parent != par
116           || n.version == Unlinked)
117         return Retry;
118       synchronized (n) {
119         prev = n.value;
120         n.value = null;
121         if (canUnlink(n)) {
122           Node c = n.left == null ? n.right : n.left;
123           if (par.left == n)
124             par.left = c;
125           else
126             par.right = c;
127           if (c != null) c.parent = par;
128           n.version = Unlinked;
129     } } }
130     fixHeightAndRebalance(par);
131   }
132   return prev;
133 }
```

Figure 2.9: Removing k's association, either by unlinking the node or by converting it to a routing node.

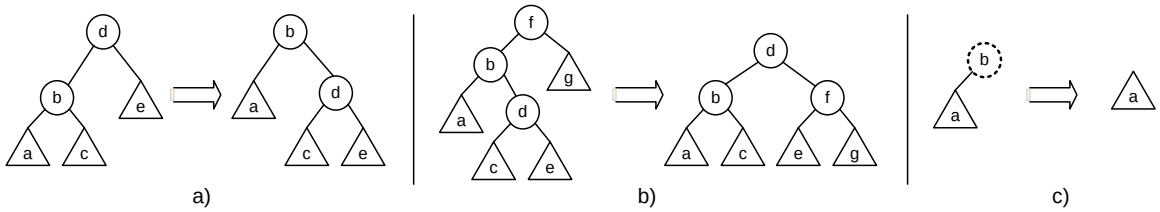


Figure 2.10: Local tree improvements: **a.** A right rotation of d. **b.** A right rotation of f over a left rotation of b. **c.** Unlink of the routing node b.

Previous research on concurrent relaxed balance trees handles this problem by using external trees [69] (or by prohibiting deletion entirely). In an external tree all key-value associations are held in leaf nodes, so there is never a deletion request that cannot be satisfied by a local operation. An external tree of size N requires $N-1$ routing nodes, increasing the storage overhead and the average search path length. We would like to avoid these penalties while still taking advantage of an external tree's simple deletion semantics.

Our solution is to use what we refer to as *partially external trees*, in which routing nodes are only created during the removal of a node with two children. Routing nodes are never created during insertion, and routing nodes with fewer than two children are unlinked during rebalancing. Removal of a node with fewer than two children is handled by immediately unlinking it. In the worst case, partially external trees may have the same number of routing nodes as an external tree, but we observe that in practice, the number of routing nodes is much smaller (see Figure 2.15). To illustrate, Figure 2.8 shows a sequence of deletions in an internal tree, an external tree, and a partially external tree.

Our tree algorithm uses the same `Node` data type to represent both key-value associations and routing nodes. This allows a value node to be converted to a routing node by modifying a field in the node, no changes to inter-node links are required. Specifically, value nodes are those with non-null `values` and routing nodes are those that have a null `value` [2]. A routing node for k is converted back to a value node by a call to `put`(k,v).

Figure 2.9 gives some of the code for implementing `remove`. The process of removal follows the same pattern as `put`. Where Line 58 of `attemptPut` performs an insertion, `attemptRemove` merely returns `null`, and where Line 62 of `attemptPut` calls

[2] In the benchmarked implementation we support user-supplied null values by encoding and decoding them as they cross the tree's public interface.

`attemptUpdate`, `attemptRemove` calls `attemptRmNode`.

`attemptRmNode` repeats our algorithm's motif of check, lock, recheck, then act. Combined with a retry loop in the caller, the motif implements optimistic concurrency control. `attemptRmNode`, however, is complicated enough to illustrate a way in which OCC tailored to an application can reduce optimistic failure rates.

Line 106 performs a check to see if the node may be unlinked or if it should be converted to a routing node. If unlinking is possible, locks are acquired on both the parent `p` and the node `n`, and then Line 121 verifies that unlinking is still possible. If unlinking is no longer possible, a generic OCC algorithm would have to roll back and retry, but this is not actually necessary. All of the locks required to convert `n` to a routing node are already held, so regardless of the outcome of this check the critical section may complete its work. In contrast, optimistic retry is required if the recheck of `canUnlink` on Line 108 shows that unlinking has become possible, because the locks held are only a subset of those required for unlinking.

2.3.6 Local tree improvements: `fixHeightAndRebalance`

The implementations of `get`, `put`, and `remove` require only a binary search tree, but to achieve good performance the tree must be approximately balanced. Our algorithm performs local improvements to the tree using the `fixHeightAndRebalance` method, which is called when a node is inserted or unlinked from the tree. This method recomputes the `height` field from the heights of the children, unlinks routing nodes with fewer than two children, and performs a single or double rotation to reduce imbalance. Our algorithm applies the same set of rotations as in a sequential AVL tree. Figure 2.10 shows a right rotation of d, a double rotation of f (we refer to this as a right-over-left rotation), and an unlink of b. The remaining possible local improvements are mirror images of these.

In a strict AVL tree the balance factor is never smaller than -2 or larger than $+2$, but in a relaxed balance AVL tree this is not the case. Multiple insertions or deletions may have accumulated before rebalancing, leading to a balance factor outside the normal range. We use the same rotation selection criteria as Bougé et al. [11]. If the apparent balance of a node n is $n.\texttt{left.height} - n.\texttt{right.height}$, then we apply a right rotation if a node's

apparent balance is $\geq +2$ and a left rotation if a node's apparent balance is ≤ -2. Prior to a right rotation of a node n, if the apparent balance of n`.left` is ≤ 0 a left rotation is first performed on n`.left`. A similar rule is applied prior to a left rotation of n.

In a concurrent relaxed balance tree there is an important distinction between the actual height and the value stored in `Node.height`. A node's `height` field records only the height as was apparent at a previous point in time, not the height that would be computed by a traversal of the tree in its current state. Bougé et al. establish the important theoretical result that, even if rebalancing is performed using only the apparent height, a sequence of localized improvements to the tree eventually results in a strict AVL tree [11]. The difficulty lies in efficiently identifying the set of nodes which can be improved.

Our algorithm guarantees that the tree will be a strict AVL tree whenever it is quiescent. Each mutating operation (insertion, removal, or rotation) is careful to guarantee that repairs will only be required for a node n or one of its ancestors, and that if no repair to n is required, no repair to any of n's ancestors is required. The node n is the one that is passed to `fixHeightAndRebalance`. The only mutation that may require repairs that don't fit this model is a double rotation in which the lower (first) rotation is not sufficient to restore balance. In this situation we do not attempt to merge the two rotations. We instead perform the lower rotation and then reapply the balance rules to the same node n.

Repair of a node requires that the children's heights be read, but performance and scalability would be heavily impacted if locks were required to perform these reads. Version numbers could be used to build a read-only atomic region, but this is not necessary. When considering if a node n should have its `height` field updated or should be rotated (both of which must be done under a lock), it is correct to perform the read of n`.left.height` and n`.right.height` without locks or optimistic retry loops. If no other thread is concurrently modifying those fields, then the check is atomic without locks or version numbers. If one of the reads is incorrect, then the thread that is performing a concurrent change is responsible for repairing n. If another thread is responsible for the repair, then it is okay if the current `fixHeightAndRebalance` incorrectly decides that no repair of n is necessary.

We omit the code for `fixHeightAndRebalance` due to space constraints (a download link for complete code is given in Appendix A), but it uses the same concurrency control structure as `get` and `remove`. An outer loop performs unlocked reads to determine whether

```
134 // n.parent, n, and n.left are locked on entry
135 void rotateRight(Node n) {
136   Node nP = n.parent;
137   Node nL = n.left;
138   Node nLR = nL.right;
139
140   n.version |= Shrinking;
141   nL.version |= Growing;
142
143   n.left = nLR;
144   nL.right = n;
145   if (nP.left == n) nP.left = nL; else nP.right = nL;
146
147   nL.parent = nP;
148   n.parent = nL;
149   if (nLR != null) nLR.parent = n;
150
151   val h = 1 + Math.max(height(nLR), height(n.right));
152   n.height = h;
153   nL.height = 1 + Math.max(height(nL.left), h);
154
155   nL.version += GrowCountIncr;
156   n.version += ShrinkCountIncr;
157 }
```

Figure 2.11: Performing a right rotation. Link update order is important for interacting with concurrent searches.

the height should be adjusted, a node should be unlinked, a rotation should be performed, or if the current node needs no repair. If a change to the node is indicated, the required locks are acquired, and then the appropriate action is recomputed. If the current locks are sufficient to perform the newly computed action, or if the missing locks can be acquired without violating the lock order, then the newly computed action is performed. Otherwise, the locks are released and the outer loop restarted. If no local changes to the tree are required then control is returned to the caller, otherwise the process is repeated on the parent.

The critical section of a right rotation is shown in Figure 2.11. This method requires that the parent, node, and left child be locked on entry. Java monitors are used for mutual exclusion between concurrent writers, while optimistic version numbers are used for concurrency control between readers and writers. This separation allows the critical region to acquire permission to perform the rotation separately from reporting to readers that

a change is in progress. This means that readers are only obstructed from Line 140 to Line 156. This code performs no allocation, has no backward branches, and all function calls are easily inlined.

2.3.7 Link update order during rotation

The order in which links are updated is important. A concurrent search may observe the tree in any of the intermediate states, and must not fail to be invalidated if it performs a traversal that leads to a branch smaller than expected. If the update on Line 145 was performed before the updates on Lines 143 and 144, then a concurrent search for `n.key` that observed only the first link change could follow a path from `n.parent` to `n.left` to `n.left.right` (none of these nodes are marked as shrinking), incorrectly failing to find `n`. In general, downward links originally pointing to a shrinking node must be changed last and downward links from a shrinking node must be changed first. A similar logic can be applied to the ordering of parent updates.

2.3.8 Iteration: `firstNode()` and `succNode(`n`)`

`firstNode()` and `succNode(`n`)` are the internal building blocks of an iterator interface. Because they return a reference to a `Node`, rather than a value, the caller is responsible for checking later that the node is still present in the tree. In an iterator this can be done by internally advancing until a non-null `value` is found.

`firstNode` returns the left-most node in the tree. It walks down the left spine using hand-over-hand optimistic validation, always choosing the left branch. Optimistic retry is only required if a node has shrunk.

`succNode` uses hand-over-hand optimistic validation to traverse the tree, but unlike searches that only move down the tree it must retry if either a shrink or grow is encountered. A complex implementation is possible that would tolerate grows while following parent links and shrinks while following child links, but it would have to perform key comparisons to determine the correct link to follow. We instead apply optimistic validation to the normal tree traversal algorithm, which is able to find the successor based entirely on the structure of the tree. If n is deleted during iteration then `succNode(`n`)` searches from the root using

```
158 static int SpinCount = 100;
159
160 void waitUntilNotChanging(Node n) {
161   long v = n.version;
162   if ((v & (Growing | Shrinking)) != 0) {
163     int i = 0;
164     while (n.version == v && i < SpinCount) ++i;
165     if (i == SpinCount) synchronized (n) { };
166   }
167 }
```

Figure 2.12: Code to wait for an obstruction to clear.

n.key.

2.3.9 Blocking readers: `waitUntilNotChanging`

Prior to changing a link that may invalidate a concurrent search or iteration, the writer sets either the `Growing` or `Shrinking` bit in the version number protecting the link, as described in Section 2.3.2. After the change is completed, a new version number is installed that does not have either of these bits set. During this interval a reader that wishes to traverse the link will be obstructed.

Our algorithm is careful to minimize the duration of the code that executes while the version has a value that can obstruct a reader. No system calls are made, no memory is allocated, and no backward branches are taken. This means that it is very likely that a small spin loop is sufficient for a reader to wait out the obstruction. Figure 2.12 shows the implementation of the waiter.

If the spin loop is not sufficient to wait out the obstruction, Line 165 acquires and then releases the changing node's monitor. The obstructing thread must hold the monitor to change `node.version`. Thus after the empty synchronized block has completed, the version number is guaranteed to have changed. The effect of a properly tuned spin loop is that readers will only fall back to the synchronization option if the obstructing thread has been suspended, which is precisely the situation in which the reader should block itself. Tolerance of high multi-threading levels requires that threads that are unable to make progress quickly block themselves using the JVM's builtin mechanisms, rather than wasting resources with fruitless retries.

2.3.10 Supporting fast `clone`

We extend our concurrent tree data structure to support `clone`, an operation that creates a new mutable concurrent tree containing the same key-value associations as the original. After a `clone`, changes made to either map do not affect the other. `clone` can be used to checkpoint the state of the map, or to provide snapshot isolation during iteration or bulk read.

We support fast cloning by sharing nodes between the original tree and the clone, lazily copying shared nodes prior to modifying them. This copy-on-write scheme requires that we be able to mark all nodes of a tree as shared without visiting them individually. This is accomplished by delaying the marking of a node until its parent is copied. All nodes in the tree may be safely shared once the root node has been explicitly marked and no mutating operations that might have not observed the root's mark are still active.

The `clone` method marks the root as shared, and then returns a new enclosing tree object with a new root holder pointing to the shared root. Nodes are explicitly marked as shared by setting their `parent` pointer to `null`. Clearing this link also prevents a Java reference chain from forming between unshared nodes under different root holders, which would prevent garbage collection of the entire original tree. Lazy copying is performed during the downward traversal of a `put` or `remove`, and during rebalancing. The first access to a child link in a mutating operation is replaced by a call to `unsharedLeft` or `unsharedRight` (see Figure 2.13). Both children are copied at once to minimize the number of times that the parent node must be locked.

Mutating operations that are already under way must be completed before the root can be marked, because they may perform updates without copying, and because they need to traverse parent pointers to rebalance the tree. To track pending operations, we separate updates into *epochs*. `clone` marks the current epoch as closed, after which new mutating operations must await the installation of a new epoch. Once all updates in the current epoch have completed, the root is marked shared and updates may resume. We implement epochs as objects that contain a count of the number of pending mutating operations, a flag that indicates when an epoch has been closed, and a condition variable used to wake up threads blocked pending the completion of a close. The count is striped across multiple cache lines

```
168 Node unsharedLeft(Node p) {
169   Node n = p.left;
170   if (n.parent != null)
171     return n;
172   lazyCopyChildren(n);
173   return p.left;
174 }
175 Node unsharedRight(Node p) { ... }
176
177 void lazyCopyChildren(Node n) {
178   synchronized (n) {
179     Node cl = n.left;
180     if (cl != null && cl.parent == null)
181       n.left = lazyCopy(cl, n);
182     Node cr = n.right;
183     if (cr != null && cr.parent == null)
184       n.right = lazyCopy(cr, n);
185   }
186 }
187 Node lazyCopy(Node c, Node newPar) {
188   return new Node(c.key, c.height, c.value, newPar,
189       0L, markShared(c.left), markShared(c.right));
190 }
191 Node markShared(Node node) {
192   if (node != null) node.parent = null;
193   return node;
194 }
```

Figure 2.13: Code for lazily marking nodes as shared and performing lazy copy-on-write. Nodes are marked as shared while copying the parent.

to avoid contention. Each snap-tree instance has its own epoch instance.

2.4 Correctness

Deadlock freedom: Our algorithm uses the tree to define allowed lock orders. A thread that holds no locks may request a lock on any node, and a thread that has already acquired one or more locks may only request a lock on one of the children of the node most recently locked. Each critical region preserves the binary search tree property, and each critical region only changes child and parent links after acquiring all of the required locks. A change of p`.left` or p`.right` to point to n requires a lock on both p, n, and the old parent of n, if any exists. This means that it is not possible for two threads T_1 and T_2 to hold locks

on nodes p_1 and p_2, respectively, and for T_1 to observe that n is a child of p_1 while T_2 observes that n is a child of p_2.

This protocol is deadlock free despite concurrent changes to the tree structure. Consider threads T_i that hold at least one lock (the only threads that may participate in a deadlock cycle). Let a_i be the lock held by T_i least recently acquired, and let z_i be the lock held by T_i most recently acquired. The node z_i is equal to a_i or is a descendent of a_i, because locks are acquired only on children of the previous z_i and each child traversal is protected by a lock held by T_i. If T_i is blocked by a lock held by T_j, the unavailable lock must be a_j. If the unavailable lock were not the first acquired by T_j then both T_i and T_j would agree on the parent and hold the parent lock, which is not possible. This means that if a deadlock cycle were possible it must consist of two or more threads $T_1 \cdots T_n$ where z_i is the parent of $a_{(i \mod n)+1}$. Because no such loop exists in the tree structure, and all parent-child relationships in the loop are protected by the lock required to make them consistent, no deadlock cycle can exist.

Linearizability: To demonstrate linearizability [47] we will define the linearization point for each operation and then show that operations for a particular key produce results consistent with sequential operations on an abstract map structure.

We define the linearization point for `put`(k,v) to be the last execution of Line 82 or 92 prior to the operation's completion. This corresponds to a successful `attemptInsert` or `attemptUpdate`. We define the linearization point for `get`(k) to be the last execution of Line 27 if Line 39 is executed, Line 34 if that line is executed and `child.value` $\neq$ `null` or `child.version` $\neq$ `Unlinked`, or otherwise Line 128 during the successful `attemptRmNode` that removed `child`. If `remove`(k) results in the execution of either Line 111 or 120 we define that to be the linearization point. We omit the details for the linearization point of a `remove` operation that does not modify the map, but it is defined analogously to `get`'s linearization point.

Atomicity and ordering is trivially provided between `put`s that linearize at Line 92 (updates) and removals that change the tree (both the introduction of routing nodes and unlinking of nodes) by their acquisition of a lock on the node and their check that `node.version` $\neq$ `Unlinked`. Nodes are marked unlinked while locked, so it is not possible for separate

threads to simultaneously lock nodes n_1 and n_2 for k, and observe that neither is unlinked. This means that any operations that operate on a locked node for k must be operating on the same node instance, so they are serialized. The only mutating operation that does not hold a lock on the node for k while linearizing is insertion, which instead holds a lock on the parent. The final hand-over-hand optimistic validation during insert (Line 80) occurs after a lock on the parent has been acquired. The validation guarantees that if a node for k is present in the map it must be in the branch rooted at `node.child(dir)`, which is observed to be empty. This means that no concurrent update or remove operation can observe a node for k to exist, and that no concurrent insert can disagree about the parent node into which the child should be inserted. Since concurrent inserts agree on the parent node, their lock on the parent serializes them (and causes the second insert to discover the node on Line 80, triggering retry).

Linearization for `get` is a bit trickier, because in some cases the last validation (Line 28) precedes the read of `child.value` (Line 34); during this interval `child` may be unlinked from the tree. If a node for k is present in the tree at Line 27 then `get` will not fail to find it, because the validation at Line 28 guarantees that the binary search invariant held while reading `node.child(dir)`. This means that when returning at Line 39 we may correctly linearize at Line 27. If a `child` is discovered that has not been unlinked prior to the read of its `value`, then the volatile read of this field is a correct linearization point with any concurrent mutating operations. If `child` $\neq$ `null` but it has been unlinked prior to Line 34 then we will definitely observe a `value` of `null`. In that case `attemptRmNode` had not cleared the child link of `node` (Line 124 or 126) when we read `child`, but it has since set `child.version` to `Unlinked` (Line 128). We therefore declare that `get` linearizes at the moment when k was definitely absent, before a potential concurrent insert of k.

2.5 Performance

In this section we evaluate the performance of our algorithm. We compare its performance to Doug Lea's lock-free `ConcurrentSkipListMap`, the fastest concurrent ordered map implementation for Java VMs of which the author is aware. We also evaluate our performance relative to two red-black tree implementations, one of which uses a single lock to

guard all accesses and one of which is made concurrent by an STM.

The benchmarked implementation of our algorithm is written in Java. For clarity this chapter describes `put`, `remove`, and `fixHeightOrRebalance` as separate methods, but the complete code does not have this clean separation. The `ConcurrentMap` operations of `put`, `putIfAbsent`, `replace`, and `remove` are all implemented using the same routine, with a case statement to determine behavior once the matching node has been found. In addition, an attempt is made to opportunistically fix the parent's `height` during insertion or removal while the parent lock is still held, which reduces the number of times that `fixHeightOrRebalance` must reacquire a lock that was just released. The benchmarked code also uses a distinguished object to stand in for a user-supplied null value, encoding and decoding at the boundary of the tree's interface.

Experiments were run on a Dell Precision T7500n with two quad-core 2.66Ghz Intel Xeon X5550 processors, and 24GB of RAM. Hyper-Threading was enabled, yielding a total of 16 hardware thread contexts. We ran our experiments in Sun's Java SE Runtime Environment, build 1.6.0_16-b01, using the HotSpot 64-Bit Server VM with default options. The operating system was Ubuntu 9.0.4 Server, with the `x86_64` Linux kernel version `2.6.28-11-server`.

Our experiments emulate the methodology used by Herlihy et al. [43]. Each pass of the test program consists of each thread performing one million randomly chosen operations on a shared concurrent map; a new map is used for each pass. To simulate a variety of workload environments, two main parameters are varied: the proportion of `put`, `remove`, and `get` operations, and the range from which the keys are selected (the "key range"). Increasing the number of mutating operations increases contention; experiments with 90% `get` operations have low contention, while those with 0% `get` operations have high contention. The key range affects both the size of the tree and the amount of contention. A larger key range results in a bigger tree, which reduces contention.

To ensure consistent and accurate results, each experiment consists of eight passes; the first four warm up the VM and the second four are timed. Throughput results are reported as operations per millisecond. Each experiment was run five times and the arithmetic average is reported as the final result.

We compare five implementations of thread-safe ordered maps:

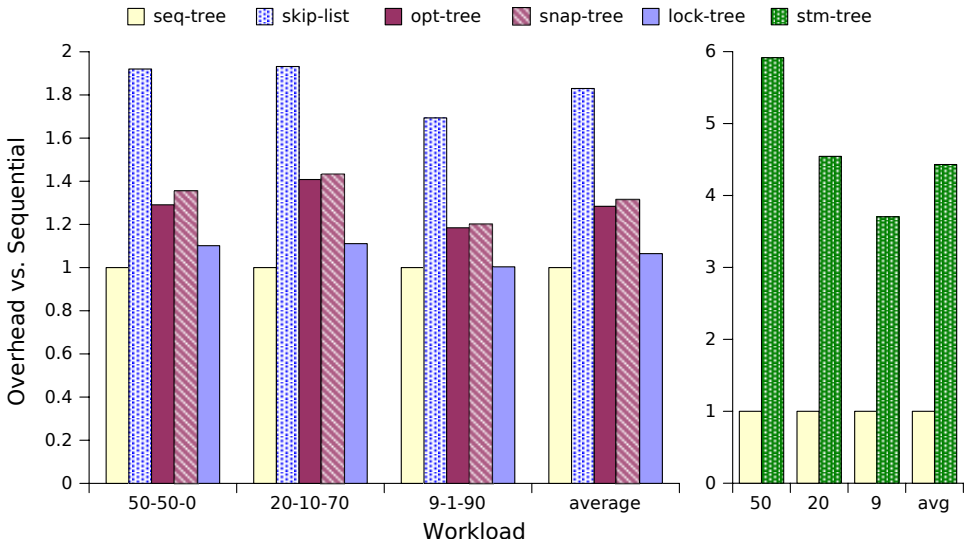


Figure 2.14: Single thread overheads imposed by support for concurrent access. Workload labels are ⟨put%⟩-⟨remove%⟩-⟨get%⟩. A key range of 2×10^5 was used for all experiments.

- **skip-list** - Doug Lea's `ConcurrentSkipListMap` This skip list is based on the work of Fraser and Harris [30]. It was first included in version 1.6 of the Java™ standard library.
- **opt-tree** - our optimistic tree algorithm.
- **snap-tree** - the extension of our algorithm that provides support for fast cloning and snapshots.
- **lock-tree** - a standard `java.util.TreeMap` wrapped by `Collections.synchronizedSortedMap()`. Iteration is protected by an explicit lock on the map.
- **stm-tree** - a red-black tree implemented in Scala[3] using CCSTM [12]. STM read and write barriers were minimized manually via common subexpression elimination. To minimize contention no size or modification count are maintained.

We first examine the single-threaded impacts of supporting concurrent execution. This is important for a data structure suitable for a wide range of uses, and it places a lower

[3]The Scala compiler emits Java bytecodes directly, which are then run on the Java VM. Scala code that does not use closures has performance almost identical to the more verbose Java equivalent.

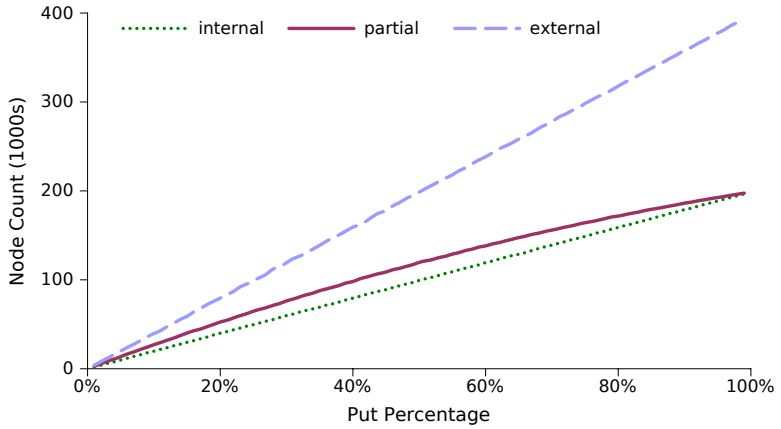


Figure 2.15: Node count as tree size increases. One million operations were performed with a varying ratio of `put` and `remove` operations, and a key range of 2×10^5; the number of nodes in the resulting tree is shown.

bound on the amount of parallelism required before scaling can lead to an overall performance improvement. We compare the sequential throughput of the five maps to that of an unsynchronized `java.util.TreeMap`, labeled "seq-tree". Values are calculated by dividing the throughput of seq-tree by that of the concurrent map. Figure 2.14 shows that on average our algorithm adds an overhead of 28%, significantly lower than the 83% overhead of skip-list, but more than the 6% imposed by an uncontended lock. The STM's performance penalty averages 443%. As expected, snap-tree is slower than opt-tree, but the difference is less than 3% for this single-threaded configuration.

Our second experiment evaluates the number of nodes present in a partially external tree compared to a fully external tree. Internal trees are the baseline, as they contain no routing nodes. To simulate a range of workloads we perform a million `put` or `remove` operations, varying the fraction of `puts` from 0% to 100%. In this experiment we use a key range of 2×10^5. The results, presented in Figure 2.15, show that partially external trees require far fewer routing nodes (on average 80% fewer) than external trees. A key range of 2×10^6 with 10 million operations yields a similar curve.

Figure 2.16 shows how throughput scales as the number of threads is swept from 1 to 64, for a range of operation mixes and various levels of contention. Moving left to right in the figure, there are fewer mutating operations and thus lower contention. Moving bottom to top, the range of keys get larger, resulting in bigger trees and lower contention. Thus

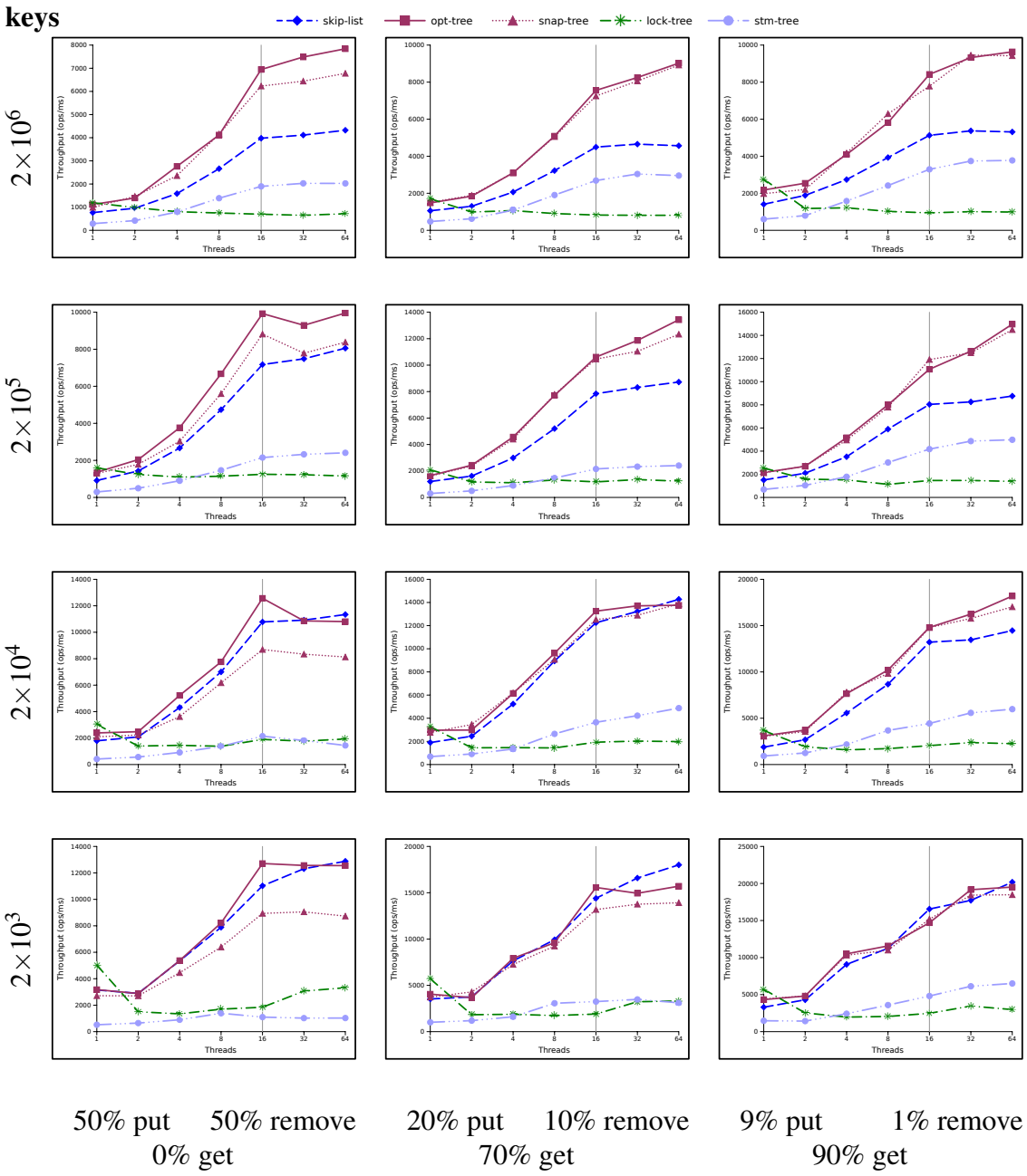


Figure 2.16: Each graph shows the throughput of the maps as thread count ranges from 1 to 64. "skip-list" is `ConcurrentSkipListMap`, "opt-tree" is our basic optimistic tree algorithm, "snap-tree" is our extension that supports fast snapshots and cloning, "lock-tree" is a synchronized `java.util.TreeMap`, and "stm-tree" is a red-black tree using STM. Moving left to right, there are fewer mutating operations and thus lower contention. Moving bottom to top, the key range get larger, resulting in bigger trees and lower contention. 16 hardware threads were available.

the lower left graph is the workload with the highest contention and the upper right is the workload with the lowest contention.

As expected, the throughput of each map, with the exception of lock-tree, generally increases as more of the system's 16 hardware thread contexts are utilized. At multiprogramming levels of 2 and 4 (32 and 64 threads) throughput flattens out. Higher numbers of threads increase the chances that a single `fixHeightAndRebalance` call can clean up for multiple mutating operations, reducing the amortized cost of rebalancing and allowing scaling to continue past the number of available hardware threads in some cases. As the key range gets smaller the absolute throughput increases, despite the higher contention, showing that both Lea's and our algorithms are tolerant of high contention scenarios. The absolute throughput also increases as the number of mutating operations decreases (going from left to right), as would be expected if reads are faster than writes. The course-grained locking of lock-tree imposes a performance penalty under any amount of contention, preventing it from scaling in any scenario. Stm-tree exhibits good scaling, especially for read-dominated configurations, but its poor single-threaded performance prevents it from being competitive with skip-list or either of our tree algorithms.

With a large key range of 2×10^6, our algorithm outperforms skip-list by up to 98%, with an average increase in throughput of 62% without fast clone and snapshot support, and 55% with such support. Both of our tree implementations continue to exhibit higher throughput with a key range of 2×10^5, but as the key range decreases and contention rises, the advantage becomes less pronounced. Opt-tree performs on par with skip-list for a key range of 2×10^4, but fails to maintain its performance advantage in multiprogramming workloads with a key range of 2×10^3, the only workload in which skip-list has noticeably higher performance. In the worst case for opt-tree (64 threads, 20-10-70 workload, and a key range of 2×10^3) it was 13% slower than skip-list. In the worst case for snap-tree (64 threads, 50-50-0 workload, and a key range of 2×10^3) it was 32% slower than skip-list. Averaged over all workloads and thread counts, opt-tree was 32% faster than skip-list and snap-tree was 24% faster than skip-list.

The primary difference between opt-tree and snap-tree is snap-tree's epoch tracking. This imposes a constant amount of extra work on each put or remove. As expected, the

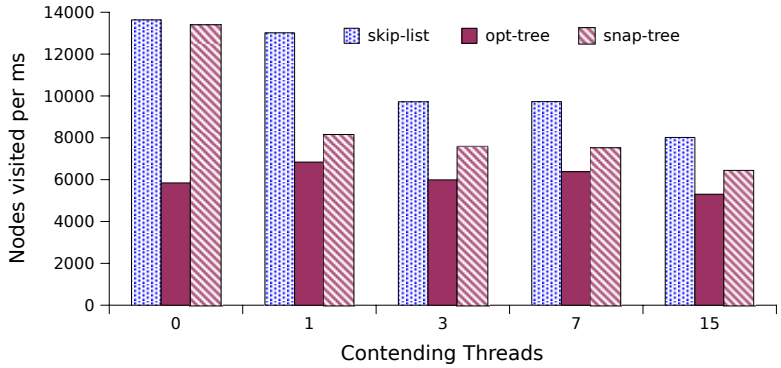


Figure 2.17: Iteration throughput of a single thread, with contending threads performing the 20-10-70 workload over 2×10^5 keys. Skip-list and opt-tree perform inconsistent iteration, snap-tree performs an iteration with snapshot isolation.

overhead of supporting snapshots decreases when moving right in the table, to configurations with fewer mutating operations. The relative cost of epoch tracking is also reduced as the tree size increases, because more work is done per operation. Across the board, snap-tree imposes a 9% overhead when compared to opt-tree, with a worst-case penalty of 31%. Snap-tree's overhead for read operations is negligible.

We next examine the performance of iterating sequentially through the map while concurrent mutating operations are being performed. Our standard per-thread workload of 20% `puts`, 10% `removes`, and 70% `gets`, and a key range of 2×10^5 is interleaved at regular intervals with a complete iteration of the map. On average only one thread is iterating at a time. We calculate throughput as the total number of nodes visited, divided by the portion of total running time spent in iteration; the results are presented in Figure 2.17. Our experimental setup did not allow us to accurately measure the execution breakdown for multiprogramming levels greater than one, so we only show results up to 16 threads. At its core, `ConcurrentSkipListMap` contains a singly-linked list, so we expect it to support very fast iteration. Iteration in opt-tree is much more complex; nevertheless, its average performance is 48% that of skip-list. No optimistic hand-over-hand optimistic validation is required to iterate the snapshot in a snap-tree, so its performance is intermediate between skip-list and opt-tree, even though it is providing snapshot consistency to the iterators.

Snap-tree provides snapshot isolation during iteration by traversing a clone of the original tree. This means that once the epoch transition triggered by `clone` has completed, `puts`

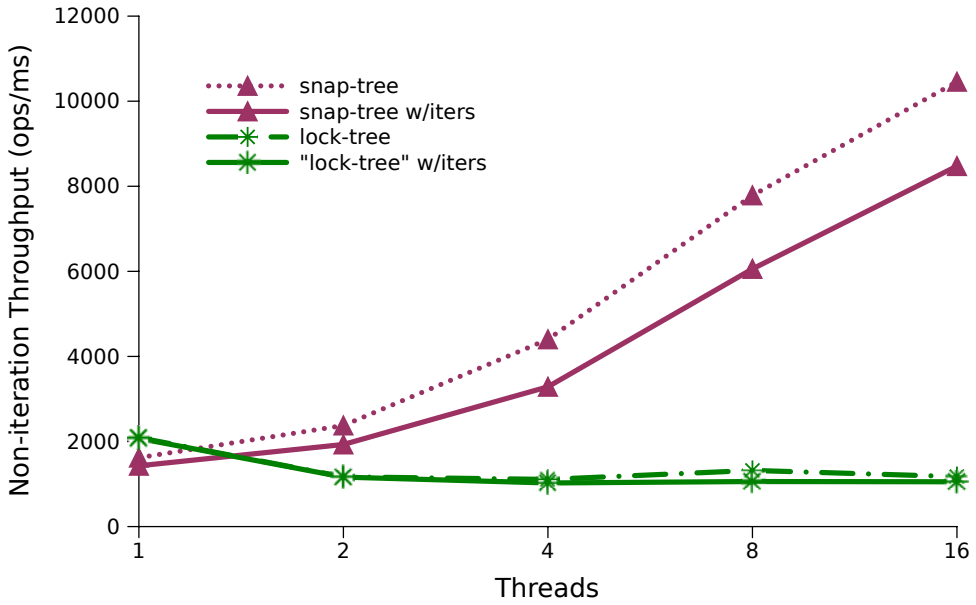


Figure 2.18: Throughput performing the 20-10-70 workload over 2×10^5 keys with concurrent consistent iterations.

and `remove`s may operate concurrently with the iteration. To evaluate the performance impact of the lazy copies requires by subsequent writes, Figure 2.18 plots the throughput of non-iteration operations during the same workload as Figure 2.17, along with the throughput with no concurrent iterations. Only snap-tree and lock-tree are shown, because they are the only implementations that allow consistent iteration. On average, concurrent iterations lower the throughput of other operations by 19% in snap-tree.

2.6 Conclusion

In this chapter we use optimistic concurrency techniques adapted from software transactional memory to develop a concurrent tree data structure. By carefully controlling the size of critical regions and taking advantage of algorithm-specific validation logic, our tree delivers high performance and good scalability while being tolerant of contention. We also explore a variation of the design that adds support for a fast clone operation and that provides snapshot isolation during iteration.

We compare our optimistic tree against a highly tuned concurrent skip list, the best performing concurrent ordered map of which we are aware. Experiments shows that our algorithm outperforms the skip list for many access patterns, with an average of 39% higher single-threaded throughput and 32% higher multi-threaded throughput. We also demonstrate that a linearizable clone operation can be provided with low overhead.

Chapter 3

Transactional Predication

3.1 Introduction

Concurrent sets and maps classes have emerged as one of the core abstractions of multi-threaded programming. They provide the programmer with the simple mental model that most method calls are linearizable, while admitting efficient and scalable implementations. Concurrent hash tables are part of the standard library of Java and C#, and are part of Intel's Thread Building Blocks for C++. Concurrent skip lists are also widely available. The efficiency and scalability of these data structures, however, comes from the use of non-composable concurrency control schemes. None of the standard concurrent hash table or skip list implementations provide composable atomicity.

Software transactional memory (STM) provides a natural model for expressing and implementing compound queries and updates of concurrent data structures. It can atomically compose multiple operations on a single collection, operations on multiple collections, and reads and writes to other shared data. Unlike lock-based synchronization, composition does not lead to deadlock or priority inversion.

If all of the loads and stores performed by a hash table, tree, or skip list are managed by an STM, then the resulting data structure automatically has linearizable methods that may be arbitrarily composed into larger transactions. The STM implementation may also provide useful properties such as: optimistic conflict detection with invisible readers

(provides the best scalability for concurrent readers on cache-coherent shared memory architectures) [79]; lock- or obstruction-freedom (limits the extent to which one thread can interfere with another) [81]; intelligent contention management (prevents starvation of individual transactions) [32]; and modular blocking using using `retry` and `orElse` (allows composition of code that performs conditional waiting) [39].

We apply the adjective 'transactional' to a data structure if its operations may participate in a transaction, regardless of the underlying implementation. The most straightforward way of implementing such an algorithm is to execute all shared memory accesses through the STM; the result will automatically be transactional, but it will suffer from high single-thread overheads and false conflicts. For many applications, trading some speed for improved programmability can be a good decision. Maps and sets are such fundamental data structures, however, that the additional internal complexity and engineering effort of bypassing the STM is justified if it leads to improvements in performance and scalability for all users.

This chapter introduces *transactional predication*, the first implementation technique for transactional maps and sets that preserves the STM's optimistic concurrency, contention management, and modular blocking features, while reducing the overheads and false conflicts that arise when the STM must mediate access to the internal structure of the collection. We factor each transactional operation into a referentially transparent lookup and a single STM-managed read or write. This separation allows the bulk of the work to bypass the STM, yet leaves the STM responsible for atomicity and isolation. Our specific contributions:

- We introduce transactional predication, the first method for performing semantic conflict detection for transactional maps and sets using an STM's structural conflict detection mechanism. This method leverages the existing research on STM implementation techniques and features, while avoiding structural conflicts and reducing the constant overheads that have plagued STM data structures (Section 3.3).

- We use transactional predication to implement transactional sets and maps on top of linearizable concurrent maps (Section 3.3). We add support for iteration in unordered maps (Section 3.5.1), and describe how to perform iteration and range-based search in

ordered maps (Section 3.5.2).

- We describe two schemes for garbage collecting predicates from the underlying map: one based on reference counting (Section 3.4.2), and one using soft references (Section 3.4.3).

- We experimentally evaluate the performance and scalability of maps implemented with transactional predication, comparing them to best-of-breed non-transactional concurrent maps, data structures implemented directly in an STM, and concurrent maps that have been transactionally boosted. We find that predicated maps outperform existing transactional maps, often significantly (Section 3.6).

3.2 Background

Sets and associative maps are fundamental data structures; they are even afforded their own syntax and semantics in many programming languages. Intuitively, concurrent sets and maps should allow accesses to disjoint elements to proceed in parallel. There is a surprising diversity in the techniques developed to deliver this parallelism. They can be roughly grouped into those that use fine-grained locking and those that use concurrency control schemes tailored to the specific data structure and its operations. Transactional predication is independent of the details of the underlying map implementation, so we omit a complete survey. We refer the reader to [46] for step-by-step derivation of several concurrent hash table and skip list algorithms.

Concurrent collection classes are widely used, but they do not provide a means to compose their atomic operations. This poses a difficulty for applications that need to simultaneously update multiple elements of a map, or coordinate updates to two maps. Consider an application that needs to concurrently maintain both a forward and reverse association between keys and values, such as a map from names to phone numbers and from phone numbers to names. If the forward and reverse maps are implemented using hash tables with fine-grained locks, then changing a phone number while maintaining data structure consistency requires acquiring one lock in the forward map (to change the value that records the phone number), and two locks in the reverse map (to remove the name from one number

and add it to another). This would require breaking the clean interface to the concurrent map by exposing its internal locks, because it is not sufficient to perform each of the three updates separately. This example also leads to deadlock if the locks are not acquired following a global lock order, which will further complicate the user's code. Lock-free hash tables don't even have the option of exposing their locks to the caller. Transactional memory, however, provides a clean model for composing the three updates required to change a phone number.

While there has been much progress in efficient execution of STM's high-level programming model, simply wrapping existing map implementations in atomic blocks will not match the performance achievable by algorithm-specific concurrency control. Data structures implemented on top of an STM face two problems:

- **False conflicts** – STMs perform conflict detection on the concrete representation of a data structure, not on its abstract state. This means that operations that happen to touch the same memory location may trigger conflict and rollback, despite the operations being semantically independent.

- **Sequential overheads** – STMs instrument all accesses to shared mutable state, which imposes a performance penalty even when only one thread is used. This penalty is a 'hole' that scalability must climb out of before a parallel speedup is observed. Sequential overheads for STM are higher than those of traditional shared-memory programming [19] and hand-rolled optimistic concurrency [14].

False conflicts between operations on a transactional data structure can be reduced or eliminated by performing *semantic conflict detection* at the level of operations. Rather than computing conflicts based on the reads from and writes to individual memory locations, higher-level knowledge is used to determine whether operations conflict. For example, adding k_1 to a set does not semantically conflict with adding k_2 if $k_1 \neq k_2$, regardless of whether those operations write to the same chain of hash buckets or rotate the same tree nodes. Because semantically independent transactions may have structural conflicts, some other concurrency control mechanism must be used to protect accesses to the underlying data structure. This means that a system that provides semantic conflict detection must break transaction isolation to communicate between active transactions. Isolation can be

relaxed for accesses to the underlying structure by performing them in open nested transactions [18, 68], or by performing them outside transactions, using a linearizable algorithm that provides its own concurrency control. The latter approach is used by transactional boosting [42].

Although semantic conflict detection using open nested transactions reduces the number of false conflicts, it exacerbates sequential overheads. Accesses still go through the STM, but additional information about the semantic operations must be recorded and shared. Semantic conflict detection using transactional boosting reduces sequential overheads by allowing loads and stores to the underlying data structure to bypass the STM entirely, but it accomplishes this by adding a layer of pessimistic two-phase locking. These locks interfere with optimistic STMs, voiding useful properties such as opacity [33], obstruction- or lock-freedom, and modular blocking [39]. In addition, boosting must be tightly integrated to the STM's contention manager to prevent starvation and livelock.

The simplicity of the underlying implementation of core library data structures is less important than their performance. Extra engineering effort expended on a transactional set or map can be amortized across many users; it is worth moving beyond the basic STM model for the internal details of the collection, so long as the simple interface is preserved.

An important additional concern that is not addressed by previous research is performance outside a transaction. Many accesses to a transactional set or map may occur outside an atomic block. Existing implementation techniques require the creation and commit of a separate transaction for each of these accesses, but this further increases the overheads of STM.

The goal of our research into transactional collections is to produce data structures whose non-transactional performance and scalability is equal to the best-of-breed concurrent collections, but that provide all of the composability and declarative concurrency benefits of STM. Transactional predication is a step in that direction.

3.3 Transactional Predication

Consider a minimal transactional set, that provides only the functions `contains(`e`)` and `add(`e`)`. Semantically, these operations conflict only when they are applied to equal elements, and at least one operation is an `add` [1]:

conflict?	`contains(`e_1`)`	`add(`e_1`)`
`contains(`e_2`)`	no	$e_1 = e_2$
`add(`e_2`)`	$e_1 = e_2$	$e_1 = e_2$

This conflict relation has the same structure as the basic reads and writes in an STM: two accesses conflict if they reference the same location and at least one of them is a write:

conflict?	`stmRead(`p_1`)`	`stmWrite(`p_1, v_1`)`
`stmRead(`p_2`)`	no	$p_1 = p_2$
`stmWrite(`p_2, v_2`)`	$p_1 = p_2$	$p_1 = p_2$

The correspondence between the conflict relations means that we can perform semantic conflict detection in our transactional set by mapping each element e to a location p, performing a read from p during `contains(`e`)`, and performing a write to p during `add(`e`)`.

Of course, conflict detection is not enough; operations must also query and update the abstract state of the set, and these accesses must be done in a transactional manner. Perhaps surprisingly, the reads and writes of p can also be used to manage the abstract state. Transactional predication is based on the observation that membership in a finite set S can be expressed as a predicate $f : U \to \{0,1\}$ over a universe $U \supseteq S$ of possible elements, where $e \in S \iff f(e)$, and that f can be represented in memory by storing $f(e)$ in the location p associated with each e. We refer to the p associated with e as that element's *predicate*. To determine if an e is in the abstract state of the set, as viewed from the current transactional context, we perform an STM-managed read of p to see if $f(e)$ is true. To add e to the set, we perform an STM-managed write of p to change the encoding for $f(e)$. The set operations are trivial as the complexity has been moved to the $e \to p$ mapping.

The final piece of `TSet` is the mapping from element to predicate, which we record

[1] We assume a non-idempotent `add` that reports set changes.

```
195 class TSet[A] {
196   def contains(elem: A): Boolean =
197       predForElem(elem).stmRead()
198   def add(elem: A): Boolean =
199       predForElem(elem).stmReadAndWrite(true)
200   def remove(elem: A): Boolean =
201       predForElem(elem).stmReadAndWrite(false)
202
203   private val predicates =
204       new ConcurrentHashMap[A,TVar[Boolean]]
205   private def predForElem(elem: A) = {
206     var pred = predicates.get(elem)
207     if (pred == null) {
208       val fresh = new TVar(false)
209       pred = predicates.putIfAbsent(elem, fresh)
210       if (pred == null) pred = fresh
211     }
212     return pred
213   }
214 }
```

Figure 3.1: A minimal but complete transactionally predicated set in Scala. Read and write barriers are explicit. *TVar* is provided natively by the STM.

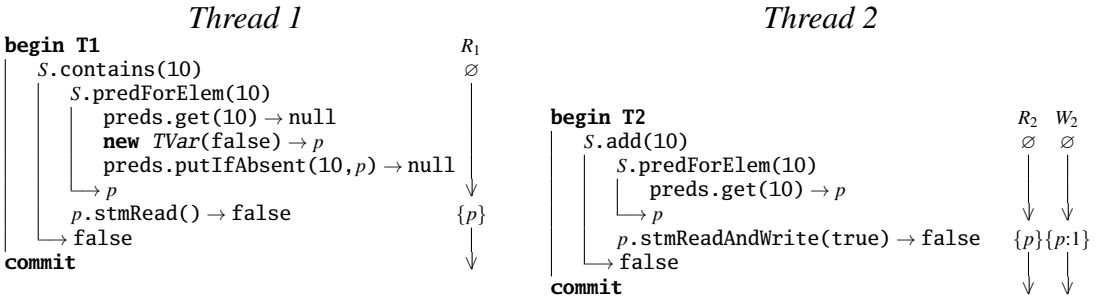


Figure 3.2: A simultaneous execution of `contains(10)` and `add(10)` using the code from Figure 3.1. R_i and W_i are the read and write sets. Thread 1 lazily initializes the predicate for element 10.

using a hash table. Precomputing the entire relation is not feasible, so we populate it lazily. The mapping for any particular e never changes, so `predForElem(elem)` is referentially transparent; its implementation can bypass the STM entirely.

Although the mapping for each element is fixed, reads and lazy initializations of the underlying hash table must be thread-safe. Any concurrent hash table implementation may be used, as long as it provides a way for threads to reach a consensus on the lazily installed key-value associations. Figure 3.1 shows the complete Scala code for a minimal transactionally predicated set, including an implementation of `predForElem` that uses `putIfAbsent` to perform the lazy initialization. `putIfAbsent(`e, p`)` associates p with e only if no previous association for e was present. It returns `null` on success, or the existing p_0 on failure. In Figure 3.1, Line 209 proposes a newly allocated predicate to be associated with `elem`, and Line 210 uses the value returned from `putIfAbsent` to compute the consensus decision.

3.3.1 Atomicity and isolation

Transactional predication factors the work of *`TSet`* operations into two parts: lookup of the appropriate predicate, and an STM-managed access to that predicate. Because the lookup is referentially transparent, atomicity and isolation are not needed. The lookup can bypass the STM completely. The read or write to the predicate requires STM-provided atomicity and isolation, but only a single access is performed and no false conflicts can result.

Bypassing the STM for the predicate map is similar to Moss' use of open nesting for *`String`*`.intern(`s`)`, which internally uses a concurrent set to merge duplicate strings [67]. Like strings interned by a failed transaction, lazily installed predicates do not need to be removed during rollback.

Figure 3.2 shows a simultaneous execution of `add(10)` and `contains(10)` using the code from Figure 3.1. Time proceeds from the top of the figure to the bottom. Because no predicate was previously present for this key, thread 1 performs the lazy initialization of the $10 \rightarrow p$ mapping. An association is present by the time that thread 2 queries the mapping, so it doesn't need to call `putIfAbsent`. At commit time, T_1's read set contains only the element p. This means that there is no conflict with a transaction that accesses any other key of S, and optimistic concurrency control can be used to improve the scalability of

parallel reads.

The abstract state of the set is completely encoded in STM-managed memory locations, so the STM provides atomicity and isolation for the data structure. Unlike previous approaches to semantic conflict detection, no write buffer or undo log separate from the STM's are required, and no additional data structures are required to detect conflicts. This has efficiency benefits, because the STM's version management and conflict detection are highly optimized. It also has semantic benefits, because opacity, closed nesting, modular blocking, and sophisticated conflict management schemes continue to work unaffected.

There are two subtleties that deserve emphasis: 1) A predicate must be inserted into the underlying map even if the key is absent. This guarantees that a semantic conflict will be generated if another transaction adds a key and commits. 2) When inserting a new predicate during `add`, the initial state of the predicate must be `false` and a transactional write must be used to set it to `true`. This guarantees that contexts that observe the predicate before the adding transaction's commit will not see the speculative add.

3.3.2 Direct STM vs. transactional predication

Figure 3.3 shows possible transactional executions of `contains(10)`. In part (a) the set is presented by a hash table with chaining. To locate the element, a transactional read must be performed to locate the current hash array, then a transactional read of the array is used to begin a search through the bucket chain. Each access through the STM incurs a performance penalty, because it must be recorded in the read set and validated during commit. In addition, reads that occur to portions of the data structure that are not specific to a particular key may lead to false conflicts. In this example, `remove(27)` will conflict with `contains(10)`, even though at a semantic level those operations are independent.

Figure 3.3b shows a predicated set executing `contains(10)`. A concurrent hash map lookup is performed outside the STM to locate the predicate. A single transactional read of the predicate is then used to answer the query. The abstract state is encoded entirely in these STM-managed memory locations; the mapping from key to predicate has no side effects and requires no atomicity or isolation. Thus no scheduling constraints are placed on the STM, and no separate undo log, write buffer or conflict information is needed.

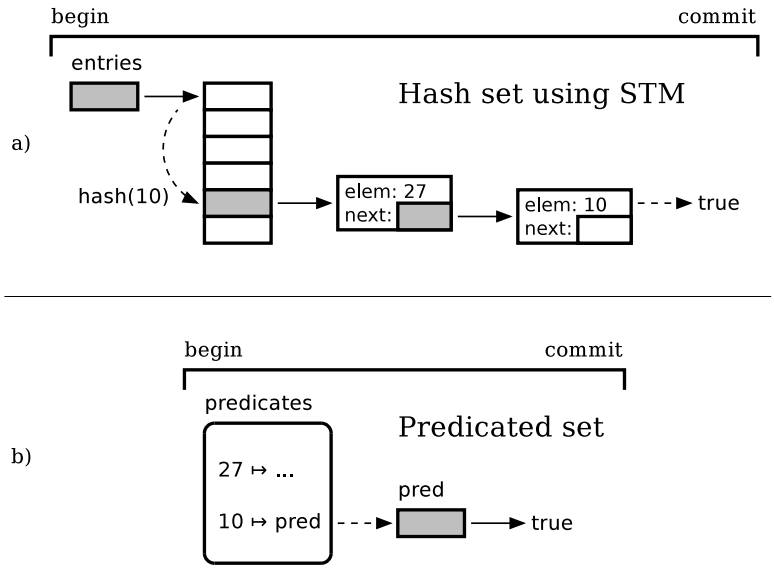


Figure 3.3: Execution of `contains(10)` in: a) a hash table performing all accesses to shared mutable state via STM; and b) a transactionally predicated set. ▭ are STM-managed reads.

3.3.3 Extending predication to maps

The predicate stores the abstract state for its associated element. The per-element state for a set consists only of presence or absence, but for a map we must also store a value. We encode this using Scala's *Option* algebraic data type, which is `Some(`v`)` for the presence of value v, or `None` for absence. *TMap*[***K***,***V***] predicates have type *TVar*[*Option*[***V***]]:

```
class TMap[K,V] {
  def get(key: K): Option[V] =
      predForKey(key).stmRead()
  def put(key: K, value: V): Option[V] =
      predForKey(key).stmReadAndWrite(Some(value))
  def remove(key: K): Option[V] =
      predForKey(key).stmReadAndWrite(None)
  private def predForKey(k: K): TVar[Option[V]] = ...
}
```

3.3.4 Sharing of uncommitted data

Like other forms of semantic conflict detection, transactional predication must make the keys of the predicate map public before the calling atomic block has committed. Carlstrom et al. [18] propose addressing this problem by using Java's `Serializable` interface to reduce keys to byte arrays before passing them across an isolation boundary in their hardware transactional memory (HTM). The version management and conflict detection in most STMs does not span multiple objects; for these systems Moss [67] shows that immutable keys can be shared across isolation boundaries. In our presentation and in our experimental evaluation we assume that keys are immutable.

3.4 Garbage Collection

The minimal implementation presented in the previous section never garbage collects its `TVar` predicates. The underlying concurrent map will contain entries for keys that were removed or that were queried but found absent. While information about absence must be kept for the duration of the accessing transaction to guarantee serializability, for a general purpose data structure it should not be retained indefinitely. Some sort of garbage collection is needed.

Predicates serve two purposes: they encode the abstract state of the set or map, and they guarantee that semantically conflicting operations will have a structural conflict. The abstract state will be unaffected if we remove a predicate that records absence, so to determine if such a predicate can be reclaimed we only need to reason about conflict detection. Semantic conflict detection requires that any two active transactions that perform a conflicting operation on the predicated collection must agree on the predicate, because the predicate's structural conflict stands in for the semantic conflict. If transaction T_1 calls `get(`k`)` and a simultaneous T_2 calls `put(`k`,`v`)`, then they must agree on k's predicate so that T_1 will be rolled back if T_2 commits.

For STMs that linearize during commit, it is sufficient that transactions agree on the predicate for k during the interval between the transactional map operation that uses k and their commit. To see that this is sufficient, let (a_i, e_i) be the interval that includes T_i's access

to k and T_i's commit. If the intervals overlap, then T_1 and T_2 agree on k's predicate. If the intervals don't overlap, then assume WLOG that $e_1 < a_2$ and that there is no intervening transaction. The predicate could not have been garbage collected unless T_1's committed state implies k is absent, so at a_2 a new empty predicate will be created. T_2's commit occurs after a_2, so T_2 linearizes after T_1. T_1's final state for k is equal to the initial abstract state for T_2, so the execution is serializable.

Algorithms that guarantee opacity can optimize read-only transactions by linearizing them before their commit, because consistency was guaranteed at the last transactional read (and all earlier ones). The TL2 [24] algorithm, for example, performs this optimization. We can provide correct execution for TL2 despite using the weaker agreement property by arranging for newly created predicates to have a larger timestamp than any reclaimed predicate. This guarantees that if a predicate modified by T_1 has been reclaimed, any successful transaction that installs a new predicate must linearize after T_1. We expect that this technique will generalize to other timestamp-based STMs. We leave for future work a formal treatment of object creation by escape actions in a timestamp-based STM. The code used in the experimental evaluation (Section 3.6) includes the TL2-specific mechanism for handling this issue.

Predicate reclamation can be easily extended to include **`retry`** and **`orElse`**, Harris et al.'s modular blocking operators [39]. All predicates accessed by a transaction awaiting **`retry`** are considered live.

3.4.1 Predicate life-cycle in a timestamp-based STM

Transactional predication constructs new *`TVar`* objects and makes them visible before the creating transaction has committed. The metadata used by a timestamp-based STM provides information about the last time at which an object was written, so we are faced with a sticky question of how to initialize a new predicate's timestamp. If we create the predicate as part of the transaction, then introduce a conflict even if all accesses to a predicate are reads. If we initialize the timestamp using the global commit timestamp, then the use of the predicate will immediately trigger either rollback or revalidation in the transaction for which it was created. If we initialize the timestamp to an older time, then the metadata is

inaccurate, which can break serializability.

Timestamp-based STM algorithms such as TL2 [24] use a per-transaction *read version* to efficiently revalidate after each transactional load. Reads are effectively projected backward in time to the moment that the read version was initialized, guaranteeing that user code is never exposed to an inconsistent state. If a memory location is encountered that has been updated since this virtual snapshot was established, the reading transaction can either roll back or advance its read version and then perform an explicit revalidation. This continuous validation has two benefits: it allows read-only transactions to skip validation during commit, and it protects the user from being exposed to inconsistent program state. The property that all transactions are consistent (although potentially out-of-date) at each point in their execution is known as opacity.

Consider a transaction A that begins execution with a read version of 1, and that then reads the predicate p_x associated with key x. A verifies that the timestamp associated with p_x is ≤ 1, and then proceeds. A transaction B then commits changes to p_x and p_y, resulting in each of those predicates having a timestamp of 2. As the last step in our problem setup, assume that p_y is garbage collected by any of the schemes that follow. At this point, if A accesses y in the transactionally predicated set, it must create a new predicate p'_y that will be immediately shared. There are three possibilities: (1) p'_y can be added to A's write set before it is shared. Writes reduce contention, have overhead, and prevent the read-only commit optimization from being used, so we would rather choose an option in which A is not considered to have written p'_y. (2) p'_y can be created with the timestamp of the most recently committed transaction. A will then include p'_y in its read set. Unfortunately, however, this strategy will require that to preserve opacity A must either roll back or revalidate itself each time that it creates a new predicate. (3) p'_y can be created with an older timestamp. In this example, however, rollback or revalidation *is* required, so an optimization that avoids it will result in an incorrect execution.

Our solution is to propagate the timestamp from p_y to p'_y. While it is not okay to pretend that p'_y was created at some point in the distant past, it is okay to pretend that it was created as part of the transaction that last wrote p_y. Clearly it would be impractical to retain exact metadata for each removed predicate, but for the case of timestamps we can merge the historical metadata of removed predicates by taking their maximum. We extend

the underlying STM with two methods:

```
def embalm(identity: Any, v: TVar[_]): Unit
def resurrect(identity: Any, v: TVar[_]): Unit
```

The `embalm` method updates the historical maximum (striped across multiple values using a hash of `identity`), and the `resurrect` method back-dates `v` to the timestamp from the most recent *TVar* embalmed with the same `identity`.

Transactional boosting manually performs pessimistic conflict detection and version management for semantic operations in the boosted data structure. While both boosting's two-phase locking and TL2's optimistic timestamp-based validation provide opacity in isolation, they do not provide opacity when combined. When a boosted transaction releases a lock during commit, it does not force the next transaction that acquires that lock to advance its read version. At each moment in a transaction, the timestamp-based reads are only guaranteed to be consistent at the moment that the read version was initialized, while the semantic operations protected by the two-phase locks are only guaranteed to be consistent at some time after the most recent lock acquisition. This also means that the read-only transaction optimization cannot be performed when boosting is used, because otherwise the transaction would not have a single linearization point.

3.4.2 Reference counting

One option for reclaiming predicates once they are no longer in use is reference counting. There is only a single level of indirection, so there are no cycles that would require a backup collector. Reference counts are incremented on access to a predicate, and decremented when the enclosing transaction commits or rolls back. Whenever the reference count drops to zero and the committed state of a predicate records absence, it may be reclaimed.

Reference counting is slightly complicated by an inability to decrement the reference count and check the value of the predicate in a single step. We solve this problem by giving present predicates a reference count bonus, so that a zero reference count guarantees that the predicate's committed state records absence. Transactions that perform a `put` that results in an insertion add the bonus during commit (actually, they just skip the normal end-of-transaction decrement), and transactions that perform a `remove` of a present key

subtract the bonus during commit. To prevent a race between a subsequent increment of the reference count and the predicate's removal from the underlying map, we never reuse a predicate after its reference count has become 0. This mechanism is appealing because it keeps the predicate map as small as possible. It requires writes to a shared memory location, however, which can limit scalability if many transactions read the same key. Reference counting can is suitable for use in an unmanaged environment if coupled with a memory reclamation strategy such as hazard pointers [64].

3.4.3 Soft references

When running in a managed environment, we can take advantage of weak references to reclaim unused predicates. Weak references can be traversed to retrieve their referent if it is still available, but do not prevent the language's GC from reclaiming the referenced object. Some platforms have multiple types of weak references, giving the programmer an opportunity to provide a hint to the GC about expected reuse. On the JVM there is a distinction between `WeakReference`s, which are garbage collected at the first opportunity, and `SoftReference`s, which survive collection if there is no memory pressure. Reclamations require mutation of the underlying predicate map, so to maximize scalability we use soft references.

Soft references to the predicates themselves are not correct, because the underlying map may hold the only reference (via the predicate) to a valid key-value association. Instead, we use a soft reference from the predicate to a discardable token object. Collection of the token triggers cleanup, so we include a strong reference to the token inside the predicate's `TVar` if the predicate indicates presence. For sets, we replace the `TVar[Boolean]` with `TVar[Token]`, representing a present entry as a `Token` and an absent entry as `null`. For maps, we replace the `TVar[Option[V]]` with a `TVar[(Token,V)]`, encoding a present key-value association as `(Token,v)` and an absent association as `(null,*)`.

If an element or association is present in any transaction context then a strong reference to the token exists. If a transactional read indicates absence, then a strong reference to the token is added to the transaction object itself, to guarantee that the token will survive at least until the end of the transaction. A predicate whose token has been garbage collected

is stale and no longer usable, the same as a predicate with a zero reference count. If a predicate is not stale, contexts may disagree about whether the entry is present or absent, but they will all agree on the transition into the stale state.

3.4.4 Optimizing non-transactional access

Ideally, transactional sets and maps would be as efficient as best-of-breed linearizable collections when used outside a transaction. If code that doesn't need STM integration doesn't pay a penalty for the existence of those features, then each portion of the program can locally make the best feature/performance tradeoff. Transactionally predicated maps do not completely match the performance of non-composable concurrent maps, but we can keep the gap small by carefully optimizing non-transactional operations.

Avoiding the overhead of a transaction: The transactionally predicated sets and maps presented so far perform exactly one access to an STM-managed memory location per operation. If an operation is called outside an atomic block, we can use an isolation barrier, an optimized code sequence that has the effect of performing a single-access transaction [49]. Our scheme for unordered enumeration (Section 3.5.1) requires two accesses for operations that change the size of the collection, but both locations are known ahead of time. Saha et al. [79] show that STMs can support a multi-word compare-and-swap with lower overheads than the equivalent dynamic transaction.

Reading without creating a predicate: While non-transactional accesses to the predicated set or map must be linearizable, the implementation is free to choose its own linearization point independent of the STM. This means that `get(`k`)` and `remove(`k`)` do not need to create a predicate for k if one does not already exist. A predicate is present whenever a key is in the committed state of the map, so if no predicate is found then `get` and `remove` can linearize at the read of the underlying map, reporting absence to the caller.

Reading from a stale predicate: `get` and `remove` can skip removal and replacement if they discover a stale predicate, by linearizing at the later of the lookup time and the time at which the predicate became stale.

Inserting a pre-populated predicate: We can linearize a `put(`k,v`)` that must insert a new predicate at the moment of insertion. Therefore we can place v in the predicate during

creation, rather than via an isolation barrier.

3.5 Iteration and Range Searches

So far we have considered only transactional operations on entries identified by a user-specified key. Maps also support useful operations over multiple elements, such as iteration, or that locate their entry without an exact key match, such as finding the smallest entry larger than a particular key in an ordered map. Transactionally predicated maps can implement these operations using the iteration or search functionality of the underlying predicate map.

3.5.1 Transactional iteration

For a transactionally predicated map M, every key present in the committed state or a speculative state is part of the underlying predicate map P. If a transaction T visits all of the keys of P, it will visit all of the keys of its transactional perspective of M, except keys added to M by an operation that starts after the iteration. If T can guarantee that no `puts` that commit before T were executed after the iteration started, it can be certain that it has visited every key that might be in M. The exact set of keys in M (and their values) can be determined by `get(`k`)` for keys k in P.

In Section 3.3 we perform semantic conflict detection for per-element operations by arranging for those operations to make conflicting accesses to STM-managed memory. We use the same strategy for detecting conflicts between insertions and iterations, by adding an insertion counter. Iterations of M read this STM-managed *TVar*[*Int*], and insertions that create a new predicate increment it. Iterations that miss a key will therefore be invalidated.

Unfortunately, a shared insertion counter introduces a false conflict between a call to `put(`k_1, v_1`)` and a call to `put(`k_2, v_2`)`. Rollbacks from this conflict could be avoided by Harris et al.'s abstract nested transactions [40], but we use a simpler scheme that stripes the counter across multiple transactionally-managed memory locations. Insertions increment only the value of their particular stripe, and iterations perform a read of all stripes. By fixing a pseudo-random binding from thread to stripe, the probability of a false conflict is

kept independent of transaction size.

There is some flexibility as to when changes to the insertion count are committed. Let t_{P+} be the time at which the key was inserted into the predicate map P and t_{M+} be the linearization time of k's insertion into M. No conflict is required for iterations that linearize before t_{M+}, because k is not part of M in their context. No conflict is required for iterations that start after t_{P+}, because they will include k. This means that any iteration that conflicts with the insertion must have read the insertion counter before t_{P+} and linearized after t_{M+}. The increment can be performed either in a transaction or via an isolation barrier, so long as it linearizes in the interval $(t_{P+}, t_{M+}]$. Incrementing the insertion counter at t_{M+}, as part of the transaction that adds k to M, allows a transactionally-consistent insertion count to be computed by summing the stripes. If a removal counter is also maintained, then we can provide a transactional `size()` as the difference.

Note that optimistic iteration is likely to produce the starving elder pathology for large maps with concurrent mutating transactions [10]. We assume that the STM's contention manager guarantees eventual completion for the iterating transaction.

3.5.2 Iteration and search in an ordered map

In an ordered map, it is more likely that an iterator will be used to access only a fraction of the elements, for example to retrieve the m smallest keys. For this use case, the insertion counter strategy is too conservative, detecting conflict even when an insertion is performed that does not conflict with the partially-consumed iterator. Ordered maps and sets also often provide operations that return the smallest or largest entry whose key falls in a range. Tracking insertions only at the collection level will lead to many false conflicts.

We solve this problem by storing an insertion count in each entry of P, as well as one additional per-collection count. An entry's counter is incremented when a predicate is added to P that becomes that entry's successor, and the per-collection counter is incremented when a new minimal entry is inserted. (Alternately a sentinal entry with a key of $-\infty$ could be used.) Because these counters only protect forward traversals, a search for the smallest key $> k$ first finds the largest key $\leq k$ and then performs a protected traversal. The successor-insertion counters for the ordered map are not useful for computing `size()`, so

we increment them using a non-transactional isolation barrier.

Despite the navigation of the underlying map required when inserting a new predicate, our scheme results in no false conflicts for `get`, `put` and `remove`, since these operations neither read nor write any of the insertion counters. Transactional iteration and range queries may experience false conflicts if a concurrent operation inserts a predicate into an interval already traversed by the transaction, but unlike false conflicts in an STM-based tree or skip list at most one interval is affected by each insertion.

3.6 Experimental Evaluation

In this section we evaluate the performance of an unordered and ordered map implemented using transactional predication. We first evaluate the predicate reclamation schemes from Section 3.4, concluding that soft references are the best all-around choice. We then show that predicated hash maps have better performance and scalability than either an STM-based hash table or a boosted concurrent hash table. Finally, we evaluate a predicated concurrent skip list.

Experiments were run on a Dell Precision T7500n with two quad-core 2.66Ghz Intel Xeon X5550 processors, and 24GB of RAM. We used the Linux kernel version `2.6.28-16-server`. Hyper-Threading was enabled, yielding a total of 16 hardware thread contexts. Code was compiled with Scala version 2.7.7. We ran our experiments in Sun's Java SE Runtime Environment, build 1.6.0_16-b01, using the HotSpot 64-Bit Server VM with compressed object pointers. We use CCSTM, a reference-based STM for Scala [16]. CCSTM uses the SwissTM algorithm [25], which is a variant of TL2 [24] that detects write-write conflicts eagerly.

Our experiments emulate the methodology used by Herlihy et al. [43]. Each pass consists of each thread performing 10^6 randomly chosen operations on a shared map; a new map is used for each pass. To simulate a variety of workloads, two parameters are varied: the proportion of `get`, `put` and `remove` operations, and the range from which the keys are uniformly selected. A smaller fraction of `gets` and a smaller key range both increase contention. Because `put` and `remove` are equally likely in our tests, the map size converges

to half the key range. To allow for HotSpot's dynamic compilation, each experiment consists of twenty passes; the first ten warm up the VM and the second ten are timed. Each experiment was run five times and the arithmetic average is reported as the final result.

3.6.1 Garbage collection strategy

To evaluate predicate reclamation strategies, Figure 3.4 shows experiments using the following map implementations:

- **conc-hash** – Lea's `ConcurrentHashMap` [59], as included in the JRE's standard library;
- **txn-pred-none** – a transactionally predicated `ConcurrentHashMap`, with no reclamation of stale predicates;
- **txn-pred-rc** – a predicated hash map that uses the reference counting scheme of Section 3.4.2; and
- **txn-pred-soft** – a predicated hash map that uses the the soft reference mechanism of Section 3.4.3.

Txn-pred-rc performs the most foreground work, but its aggressive reclamation yields the smallest memory footprint. Txn-pred-soft delays predicate cleanup and has larger predicate objects than txn-pred-rc, reducing locality. Because the performance effect of locality depends on the working set size, we show a sweep of the key range for a fixed instruction mix (80% get, 10% put and 10% remove), at minimum and maximum thread counts. The optimizations from Section 3.4.4 also have a large impact, so we show both non-transactional access and access in transactions that perform 64 operations (txn2's curves are similar to txn64's).

Except for conc-hash, the non-txn experiments represent the performance of a map that *supports* transactional access, but is currently being accessed outside an atomic block. Conc-hash is faster than any of the transactional maps, at least for 1 thread. For some multi-thread non-txn experiments, however, txn-pred-none and txn-pred-soft perform better than conc-hash, despite using a `ConcurrentHashMap` in their implementation. This is because they allow a predicate to remain after its key is removed from the abstract state of the

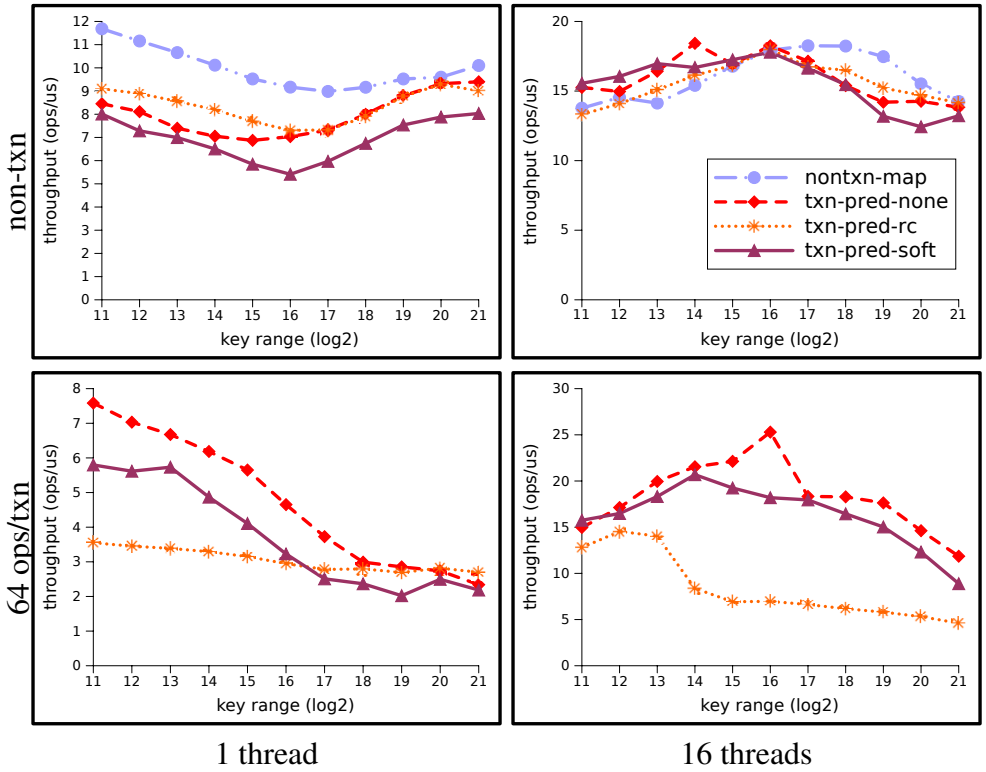


Figure 3.4: Throughput for three predicate reclamation strategies (none, reference counting and soft references), with 80% reads. Lea's non-composable *`ConcurrentHashMap`* is included for reference.

map, replacing a use of conc-hash's contended segment lock with an uncontended write to a `TVar`. If the key is re-added, the savings are doubled. This effect appears even more prominently in Figures 3.5 and 3.6, discussed below.

For most transactional configurations (that cannot use the non-transactional optimizations) txn-pred-soft is both faster and more scalable than txn-pred-rc. The exception is uncontended (1 thread) access to a large map, where reference counting's smaller memory footprint has a locality advantage that eventually compensates for its extra work. The largest difference in memory usage between txn-pred-soft and txn-pred-rc occurs for workloads that perform transactional `get` on an empty map for many different keys. In this case txn-pred-rc's predicate map will contain only a few entries, while txn-pred-soft's may grow quite large. For single-threaded access and 2^{21} key range, this 0% hit rate scenario yields

73% higher throughput for reference counting. Once multiple threads are involved, however, reference counting's locality advantage is negated by shared writes to the underlying predicate map, and txn-pred-soft performs better across all key ranges and hit rates.

Txn-pred-soft shows better overall performance than txn-pred-rc, so it is our default choice. For the rest of the performance evaluation we focus only on txn-pred-soft.

3.6.2 Comparison to other transactional maps

Figure 3.5 compares the performance of txn-pred-soft to a hash table implemented via STM, and to a transactionally boosted map. Conc-hash is included in the non-txn configurations for reference:

- **stm-hash** – a hash map with 16 segments, each of which is a resizeable transactional hash table.
- **boosting-soft** – a transactionally boosted *`ConcurrentHashMap`*. Soft references are used to reclaim the locks.

An obvious feature of most of the graphs is decreasing or constant throughput when moving from 1 to 2 threads. This is a consequence of the Linux scheduling policy, which prefers to spread threads across chips. This policy maximizes the cache and memory bandwidth available to an application, but it increases coherence costs for writes to shared memory locations. We verified this by repeating experiments from Figure 3.5 using a single processor. For very high-contention experiments such as $\langle \text{non-txn}, 2^{11}, 0\% \text{ get} \rangle$, off-chip coherence costs outweigh the benefits of additional threads, yielding higher throughput for 8 threads on 1 chip than 16 threads on 2 chips.

Stm-hash includes several optimizations over the hash table example used in Section 3.3.2. To reduce conflicts from maintaining load factor information, stm-hash distributes its entries over 16 segments. Each segment is an independently resizeable hash table. In addition, segments avoid unnecessary rollbacks by updating their load factor information in an abstract nested transaction (ANT) [40]. To reduce the number of transactional reads and writes, bucket chains are immutable. This requires extra object allocation during `put` and `remove`, but improves both performance and scalability. Finally, we optimized non-transactional `get` by performing its reads in a hand-rolled optimistic retry loop,

avoiding the overheads of transaction setup and commit.

The optimizations applied to stm-hash help it to achieve good performance and scalability for read-dominated non-txn workloads, and read- or write-dominated workloads with few accesses. Non-txn writes have good scalability, but their single-thread performance is poor; the constant overheads of the required transaction can't be amortized across multiple operations. Stm-hash has good single-thread performance when used by transactions that perform many accesses, but does not scale well in that situation. Each transaction updates several segments' load factors, making conflicts likely. Although the ANTs avoid rollback when this occurs, conflicting transactions cannot commit in parallel.

Txn-pred-soft is faster than boosting-soft for every configuration we tested. For non-txn workloads, predication has two advantages over boosting: 1) The optimizations of Section 3.4.4 mean that txn-pred's non-transactional `get(`k`)` never needs to insert a predicate, while boosting must insert a lock for k even if k is not in the map. This effect is visible in the non-txn 80% get configurations across all thread counts. 2) Boosting's scalability is bounded by the underlying *`ConcurrentHashMap`*. For write-heavy workloads conc-hash's 16 segment locks ping-pong from core to core during each insertion or removal. Txn-pred-soft's predicates are often retained (and possibly reused) after a key is removed from the map, moving writes to lightly-contended *`TVar`*s. Conc-hash's bound on boosting can be clearly seen in $\langle \text{non-txn}, 2^{11}, 0\% \text{ get} \rangle$, but applies to all workloads.

Some of predication's performance advantage across all thread counts comes from a reduction in single-thread overheads. Boosting's implementation incurs a per-transaction cost because each transaction that accesses a boosted map must allocate a side data structure to record locks and undo information, and a commit and rollback handler must be registered and invoked. Txn-pred-soft uses neither per-transaction data or transaction lifecycle callbacks. Small transactions have less opportunity to amortize boosting's overhead, so the single-thread performance advantage of predication is higher for txn2 configurations than for txn64.

The remainder of predication's performance advantage is from better scaling, a result of its use of optimistic reads and its lack of interference with the STM's contention management strategy. The scaling advantage of optimistic reads is largest for small key ranges and long transactions, both of which increase the chance that multiple transactions

will be accessing a map entry; see $\langle 64 \text{ ops/txn}, 2^{11} \text{ keys}, 80\% \text{ get}\rangle$. Boosting's locks are not visible to or revocable by the STM's contention manager, so they negate its ability to prioritize transactions. This is most detrimental under high contention, such as $\langle 64 \text{ ops/txn}, 2^{11} \text{ keys}, 0\% \text{ get}\rangle$. In this experiment boosting achieves its best throughput at 1 thread, while CCSTM's contention manager is able to provide some scalability for predication despite a large number of conflicts.

3.6.3 Ordered maps

Finally, we evaluate the performance of a transactionally predicated ordered map, which adds optimistic ordered iteration and range searches to the key-equality operations. The underlying predicate map is Lea's *`ConcurrentSkipListMap`*. Figure 3.7 compares the performance of the predicated skip list to a red-black tree implemented using STM. (We also evaluated an STM skip list, but it was slower than the red-black tree.) The predicated ordered map outperforms the STM-based ordered map for both configurations and across all thread counts.

3.7 Related Work

3.7.1 Avoiding structural conflicts

Herlihy et al. introduced early release as a method to reduce the chance of structural conflict during tree searches in their seminal paper on dynamically-sized STM [44]. Early release allows the programmer to remove elements from a transaction's read set if it can be proved that the results of the transaction will be correct regardless of whether that read was consistent. This reasoning is subtle, especially when reasoning about STM as a means for composing operations, rather than an internal data structure mechanism for implementing linearizability. Felber et al.'s elastic transactions provide the conflict reduction benefits of early release with a more disciplined model [29]. Neither of these techniques reduces the number of transactional barriers.

Moss [67] describes using open nested transactions to eliminate conflicts arising from *`String`*`.intern(`*s*`)`, which uses a globally shared hash table to merge duplicates. Calls to

`intern` never semantically conflict and need not be rolled back.

Harris et al.'s abstract nested transactions (ANT) [40] allow portions of a transaction to be retried, increasing the number of transactions that can commit. ANTs could be used to insulate the caller's transaction from false conflicts that occur inside data structure operations. However, they do not avoid the need to roll back and retry the nested transaction, and add extra overheads to the base sequential case.

Another way to reduce structural conflicts is to use an algorithm that allows bookkeeping work such as tree rebalancing to be performed separately from semantic changes. Ballard showed that the use of a relaxed balance tree algorithm can improve scalability in an STM [4]. The total amount of work is not reduced, however, so single-thread overheads remain high.

3.7.2 Semantic conflict detection

Semantic conflict detection using open nested transactions was described concurrently by Ni et al. [68] and Carlstrom et al. [18]. Ni et al. use open nested transactions in an STM to commit updates to transactionally managed data structures before their enclosing transaction commits, by tracking semantic conflicts with pessimistic locks. Their locks support shared, exclusive, and intension exclusive access, which enables them to support concurrent iteration or concurrent mutation, while correctly preventing simultaneous mutation and iteration. Carlstrom et al. use open nested transactions in a hardware transactional memory (HTM) to manage both the shared collection class and information about the operations performed by active transactions. This side information allows optimistic conflict detection. It is more general in form than abstract locks, and provides better fidelity than locks for range queries and partial iteration. Approaches that use open nesting still use transactions to perform all accesses to the underlying data structure, and incur additional overhead due to the side data structures and deeper nesting. This means that although they reduce false conflicts, they don't reduce STM's constant factors.

Kulkarni et al. associate lists of uncommitted operations with the objects in their Galois system [54], allowing additional operations to be added to a list only if there is no semantic conflict with the previous ones. Several application-specific optimizations are used to

reduce overheads.

Herlihy et al. described transactional boosting [42], which addresses both false conflicts and STM constant factors. Boosting uses two-phase locking to prohibit conflicting accesses to an underlying linearizable data structure. These locks essentially implement a pessimistic visible-reader STM on top of the base STM, requiring a separate undo log and deadlock avoidance strategy. The resulting hybrid provides atomicity and isolation, but loses useful properties and features of the underlying STM, including starvation freedom for individual transactions, obstruction- or lock-freedom, modular blocking, and timestamp-based opacity. In addition, boosting requires that the STM linearize during commit, which eliminates the read-only transaction optimization possible in STMs such as TL2 [24] and SwissTM [25].

3.7.3 Serializing conflicting transactions

Ramadan et al. adapt ideas from thread-level speculation in their dependence-aware transactional memory (DATM) [75]. This technique constrains the commit order when conflicts are detected, and then speculative forwards values from earlier transactions to later ones. This reduces the penalty normally imposed by false conflicts by allowing the transactions to commit anyway. Transactional predication relies only on serializability, so predicated data structures executed in an STM with DATM would allow even transactions with true semantic conflicts to be successfully serialized. In contrast, transactional boosting requires that forwarding be reimplemented at the boosting level, as in Koskinen et al's concurrent non-commutative boosted transactions [53].

3.8 Conclusion

This chapter has introduced transactional predication, a technique for implementing high performance concurrent collections whose operations may be composed using STM. We have shown that for sets and maps we can choose a representation that allows a portion of the transactional work to safely bypass the STM. The resulting data structures approximate semantic conflict detection using the STM's structural conflict detection mechanism, while

leaving the STM completely responsible for atomicity and isolation. Predication is applicable to unordered and ordered sets and maps, and can support optimistic iteration and range queries.

Users currently face a tradeoff between the performance of non-composable concurrent collections and the programmability of STM's atomic blocks; transactional predication can provide both. Predicated collections are faster than existing transactional implementations across a wide range of workloads, offer good performance when used outside a transaction, and do not interfere with the underlying STM's opacity, modular blocking or contention management.

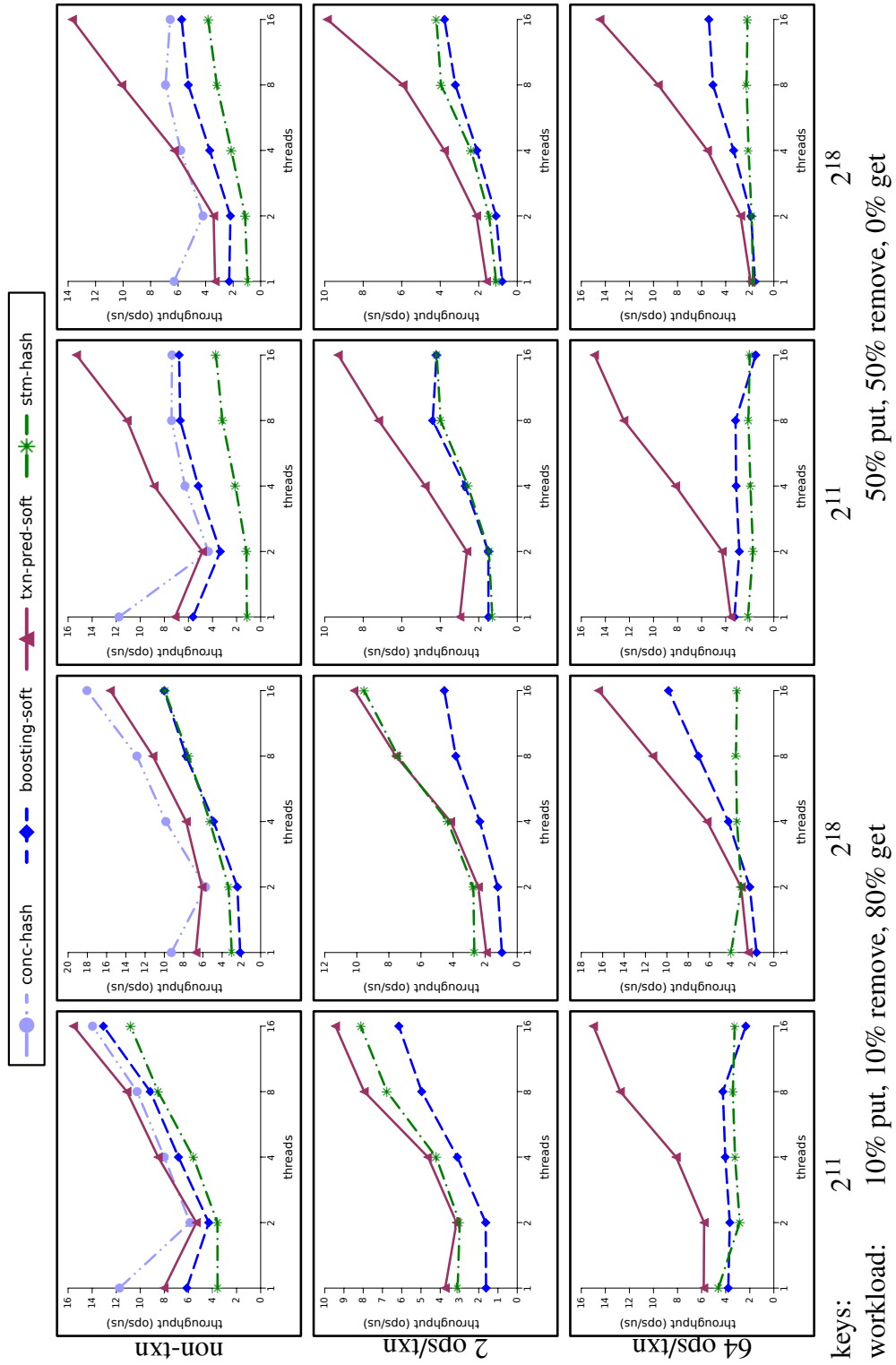
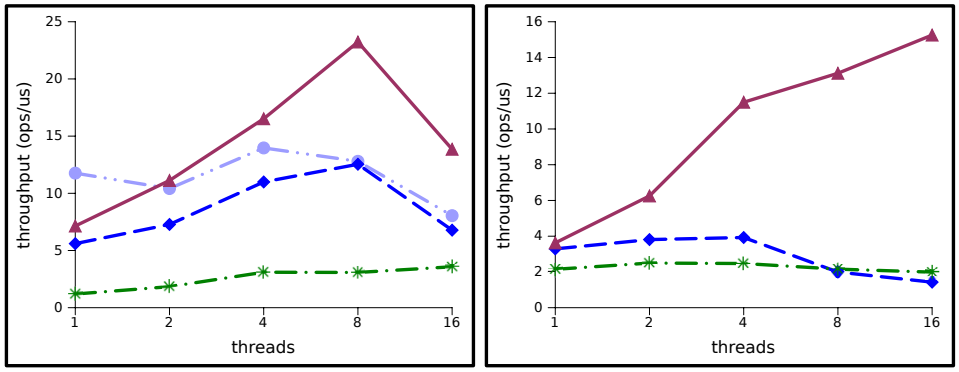
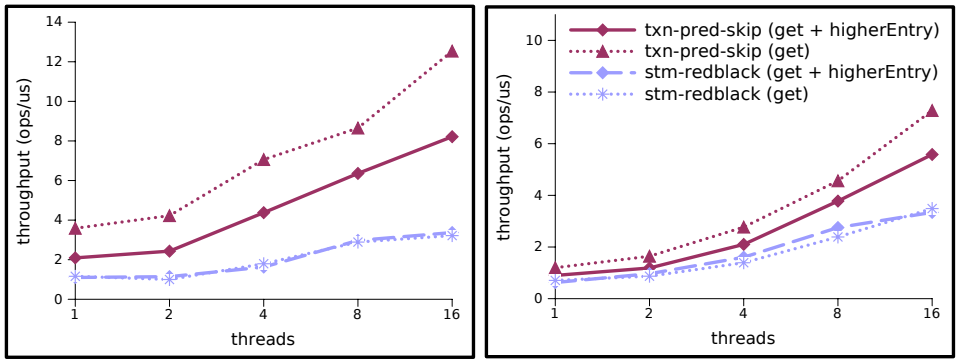


Figure 3.5: Throughput of transactional map implementations across a range of configurations. Each graph plots operations per microsecond, for thread counts from 1 to 16.



non-txn 64 ops/txn

Figure 3.6: Throughput for Figure 3.5's $\langle$non-txn, 2^{11} keys, 0% get$\rangle$ and $\langle$txn64, 2^{11} keys, 0% get$\rangle$ experiments, with a non-default scheduling policy that uses one chip for thread counts ≤ 8.



2^{11} keys 2^{18} keys

Figure 3.7: Throughput for ordered transactional maps performing 80% reads, either all `get` or half `get` and `higherEntry`. *`SortedMap`*`.higherEntry(`k`)` returns the entry with the smallest key $> k$.

Chapter 4

Transactional Maps with Snapshots

4.1 Introduction

In Chapter 2 we used optimistic concurrency control and lazy copy-on-write to efficiently provide snapshots and consistent iteration of a linearizable map. SnapTree, the resulting data structure, allows readers to traverse a consistent version of the tree without blocking subsequent writers and without any large-latency operations. While SnapTree's `clone` operation is powerful, SnapTree does not provide arbitrary composability. In particular, there is no provision for composing mutating operations or for performing an atomic snapshot across multiple trees.

In Chapter 3 we introduced transactional predication, which reduces the performance overhead of STM by bypassing the STM for a substantial fraction of the work required in a transactional map. By reducing the number of STM-managed accesses, transactional operations are accelerated and non-transactional operations can use highly-optimized isolation barriers rather than dynamically-sized transactions. Transactionally predicated maps have strengths that complement those of SnapTrees: predicated maps support composition of reads and writes, but they do not support an efficient `clone` or consistent non-transactional iteration.

In this chapter we combine the disparate strategies from Chapers 2 and 3. We start

	SnapTree	predicated hash table	transactional hash trie
linearizable read of $m(k)$	✓	✓	✓
linearizable write of $m(k)$	✓	✓	✓
compare-and-swap of $m(k)$	✓	✓	✓
atomically read $m(k_1)$ and $m(k_2)$	✓	✓	✓
atomically read $m_1(k)$ and $m_2(k)$		✓	✓
fast atomic m`.clone`	✓		✓
consistent iteration inside transaction	✓	✓	✓
consistent iteration outside transaction	✓		✓
atomically compose reads and writes		✓	✓
full STM integration		✓	✓

Figure 4.1: Features for the data structures developed in Chapters 2, 3 and 4.

with a lock-free concurrent hash trie that supports snapshots, and then implement the algorithm by using STM-managed references rather than primitive compare-and-swap operations. The hash trie can be used as a linearizable map outside a transaction, but when it is executed inside a transaction its linearizable operations compose. Figure 4.1 shows the features provided by the transactional hash trie, compared to SnapTrie and a transactionally predicated map.

Surprisingly, we found that executing a linearizable lock-free algorithm inside a transaction is not sufficient to guarantee atomicity and isolation if the algorithm is also used outside a transaction, even for strongly atomic TMs. Lock-free operations whose linearization point may be another thread's memory access can appear to be interleaved with a transaction; serializability of the underlying memory accesses is not sufficient to guarantee atomicity and isolation of the operations on the abstract state of the linearizable object.

Our lock-free hash trie can be correctly composed using memory transactions, because all of its linearization points occur at memory accesses of the current thread. In Section 4.3 we discuss the interaction of lock-freedom and memory transactions, and we briefly introduce *internally linearizable* objects that can safely be accessed directly and composed using transactions.

The result of tailoring our concurrent map and set implementation to the STM environment is an unordered concurrent map that adds snapshots and arbitrary composability,

with a low performance penalty. We explored the usefulness of the resulting model by rewriting the lock manager of the Apache Derby database. The lock manager is a complicated piece of concurrent code that is critical to the performance and scalability of the database. In 2007 scalability limitations forced a rewrite using fine-grained locking and Lea's `ConcurrentHashMap`. We reimplemented the lock manager and deadlock detection by directly encoding the underlying relations as set of transactional maps. Our implementation scales as well as the fine-grained locking solution and delivers 98% of its peak performance, while using dramatically less code.

Our specific contributions:

- We describe a novel lock-free mutable hash trie that supports efficient `clone` via copy-on-write. This algorithm can be used directly on hardware that supports DCAS, or it can use a software DCAS construction (Section 4.2).
- We examine the conditions under which a linearizable data structure may be concurrently accessed from both inside and outside a transaction when its shared memory accesses are made through an STM. We show that, surprisingly, lock-free algorithms can fail to behave as expected even for strongly atomic STMs (Section 4.3).
- We use the mutable hash trie to implement an unordered transactional map that supports a copy-on-write `clone` operation. This data structure provides the snapshot iteration benefits of SnapTree while approaching the peak performance of a transactionally predicated concurrent hash map (Section 4.4).
- We directly compare the performance of our transactional hash trie to the transactional maps from Chapter 3, using the same microbenchmarks used to evaluate transactional predication. We find that neither implementation has a clear performance advantage and neither implementation has a performance pathology, despite the trie's extra functionality (Section 4.6.2).
- We evaluate the transactional hash trie in a large application by rewriting Apache Derby's database lock manager, a sophisticated concurrent component that uses fine-grained

locking. Our code using STM is substantially shorter and simpler than the original, yet it scales as well and achieves 98% of the original's peak performance (Section 4.6.4).

4.2 A Lock-Free Copy-On-Write Hash Trie

In this section we will describe a lock-free algorithm for implementing an unordered map data structure that also provides an $O(1)$ linearizable `clone` operation. The algorithm makes use of a restricted double-compare single-swap (RDCSS) primitive, which is not generally available in hardware[1], but Harris et al.'s lock-free RDCSS construction can be used for machines that support only CAS [41]. Our final goal is an algorithm that provides STM integration, so we will implement RDCSS using STM primitives.

4.2.1 Hash tries

A tree-like structure is required to implement snapshots that perform an incremental copy-on-write, but there is no natural ordering available for the keys of an unordered map. The keys provide only equality comparison (`equals`) and hash code computation (`hashCode`). This means that we must construct the tree using the key's hash code.

A trie is a tree that performs lookups on keys made from a sequence of symbols [31]. The branching factor of each level of the tree matches the number of possible symbols. To locate a key starting with the symbol s_i the i-th child of the root is chosen, and then the process is repeated with the remainder of the key.

Hash codes are (assumed to be) uniformly randomly distributed[2], so Devroye's results prove that the expected leaf depth of a bitwise trie of hash codes is $O(\log n)$ [23]. A hash trie with branching factor BF breaks a key's hash code into groups of $\log_2 BF$ bits, and uses the groups as the symbols in a trie lookup. The original key–value associations are stored in the leaves of the hash trie; key equality is checked only after the hash code of the key has been used to locate an association.

[1] RDCSS is equivalent to DCAS if the old and new values of the first address are restricted to be identical and the address space of the two arguments are distinct.

[2] We scramble and XOR the bits of the hash function so that this assumption holds even if only some of the bits are well distributed.

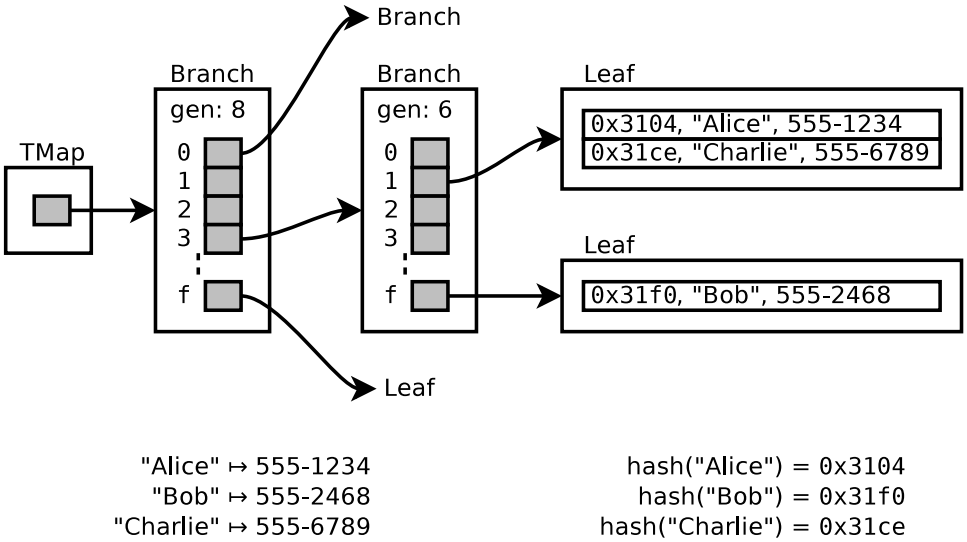


Figure 4.2: The hash trie data structure. ▭ are mutable references.

A practical efficiency concern with tries is space wasted by unused children. Hash array mapped tries address this inefficiency by compacting the child pointers, and then using a bitmap to locate the packed index that corresponds to a symbol [3]. This poses complications for a concurrent data structure, as it leads to conflicts for operations that access separate children. We notice that the vast majority of the wasted space occurs only near the leaves, so it is not necessary to compact the upper levels of the tree.

Our data structure must deal with hash collisions, which result in multiple key–value associations in the same leaf. Our solution to this problem also helps minimize wasted space due to unused branches. We let each leaf instance hold an array of up to LC triples of $\langle h, k, v \rangle$, where LC is the leaf capacity, h is the key's hash code, k is the key and v is the associated value. (Leaves can become larger than LC if there is an exceptionaly large number of collections, because every key with the same hash code must reside in the same leaf.) The triples are stored by increasing h. Maintaining this array is fast in practice because it has good cache locality and minimizes per-object overheads.

We avoid the concurrency problems of packed arrays in the leaves by making them immutable. The leaf is copied and repacked during insertion and deletion of key–value associations. The leaf is also copied during updates, so that all leaf changes are reflected in the same memory location (the leaf's parent's child reference). The unordered map exposed

to the outside world contains a reference to either a branch or a leaf. The multi-element leaf means that small maps consist of just the map instance and a leaf. Figure 4.2 shows a hash trie, and Figure 4.3 shows the signatures of the map, branch and leaf types.

4.2.2 Generation counts

After a call to `clone`, the existing nodes of the hash trie must no longer be modified. We can't mark each of the nodes immediately, because that would require that the entire tree be traversed. Instead, we record a generation number in each mutable node, and then we detect frozen nodes by checking if a node's generation number differs from that of the root branch. While the following function is never actually used in the trie's implementation, we could can compute if a node is frozen with

```
def isFrozen(b: Node[_, _]) = b.gen != rootRef.get.gen
```

We freeze a node when it is accessible from multiple `LockFreeHashTrie` instances. Unfrozen nodes may be accessed from multiple threads.

Using generations to detect sharing is different than the lazy marking scheme we employed in Chapter 2's SnapTrie. Generations have the benefit that `clone` can be run concurrent with attempts to modify the tree, there is no need to separate updates into epochs (Section 2.3.10). SnapTree needs the lazy marking scheme to clear its nodes' parent pointers, which are required for bottom-up relaxed rebalancing. Because no rebalancing is necessary for hash tries there is no need for parent pointers, so we can use the simpler and more cache friendly generation mechanism.

Each child reference from a branch b can point to a leaf, a branch with a generation older than b's, or a branch from the same generation as b. Because leaves are immutable they don't need to store generation. To simplify the code we assign all leaves a generation of -1. The lifecycle of a reference is shown in Figure 4.4. Note that the child branch cannot be replaced once it has the same generation as the parent, and a child branch from an old generation can be replaced at most once. Freezing occurs by increasing the root generation rather than be decreasing the generation of any of the child branches. We use a 64-bit generation value to avoid any risk of overflow.

```
215 class Ref[A](init: A) {
216   def get: A
217   def set(v: A)
218   def cas(v0: A, v1: A): Boolean
219 }
220 object Ref {
221   def rdcss[A, B](a: Ref[A], a0: A, b: Ref[B], b0: B, b1: B): Boolean
222 }
223
224 class LockFreeHashTrie[A, B](contents: Node[A, B]) {
225   val rootRef = new Ref(contents)
226   def get(key: A): Option[B]
227   def put(key: A, value: B): Option[B]
228   def remove(key: A): Option[B]
229   def clone: LockFreeHashTrie[A, B]
230 }
231
232 abstract class Node[A, B] {
233   def gen: Long
234 }
235
236 class Leaf[A, B](
237     val hashes: Array[Int],
238     val keys: Array[A],
239     val values: Array[B]) extends Node[A, B] {
240   def gen = -1L
241   def get(key: A, hash: Int): Option[B]
242   def withPut(key: A, hash: Int, value: B): Node[A, B]
243   def withRemove(key: A, hash: Int): Leaf[A, B]
244 }
245 object Leaf {
246   def empty[A, B]: Leaf[A, B]
247 }
248
249 class Branch[A, B](
250     val gen: Long,
251     val childRefs: Array[Ref[Node[A, B]]]) extends Node[A, B] {
252   def copy(newGen: Long): Branch[A, B]
253 }
```

Figure 4.3: Class signatures for the lock-free hash trie. *Ref* is a mutable cell that supports CAS and RDCSS. Members of a Scala `object` behave like static members of the corresponding class.

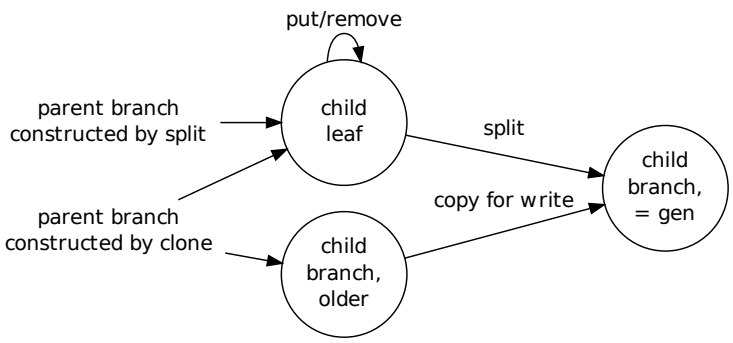


Figure 4.4: The lifecycle of each child reference in a branch.

```
class LockFreeHashTrie[A, B](contents: Node[A, B]) {

  def clone = new LockFreeHashTrie(cloneContents())

  @tailrec private def cloneContents(): Node[A, B] = {
    rootRef.get match {
      case leaf: Leaf[A, B] => leaf // easy
      case branch: Branch[A, B] => {
        if (rootRef.cas(branch, branch.copy(branch.gen + 1)))
          branch.copy(branch.gen + 1) // CAS was successful
        else
          cloneContents() // CAS failed, retry
      }
    }
  }
}
```

Figure 4.5: Code for lock-free hash trie `clone`.

The *`Branch`* instances of a trie that are referenced by exactly one *`LockFreeHashTrie`* instance are exactly those that have the same generation number as the root branch, if any. This means that we can cause the branches to be considered shared by installing a new root branch using CAS. The cloned trie uses a separate copy of the root branch with the same new generation number, but because the only nodes that are shared between tries are immutable or have old generations, there is no problem with the cloned trie using the same new generation. If a trie consists of only a single *`Leaf`* then nothing needs to be done, because leaves are immutable. Figure 4.5 shows the resulting implementation of `clone`.

```
270 class LockFreeHashTrie[A, B] ... {
271   private def LogBF = 4
272   private def BF = 1 << LogBF
273   private def chIndex(shift: Int, hash: Int) = (hash >>> shift) & (BF - 1)
274
275   def get(key: A): Option[B] = get(rootRef.get, rootRef, 0, key, key.##)
276
277   @tailrec private def get(root: Node[A, B],
278                            nodeRef: Ref[Node[A, B]],
279                            shift: Int,
280                            key: A,
281                            hash: Int): Option[B] = {
282     (if (shift == 0) root else nodeRef.get) match {
283       case leaf: Leaf[A, B] => {
284         if (shift != 0 && (rootRef.get != root))
285           get(rootRef.get, rootRef, 0, key, hash) // retry from root
286         else
287           leaf.get(key, hash)
288       }
289       case branch: Branch[A, B] => get(
290           root, branch.childRefs(chIndex(shift, hash)), shift + LogBF,
291           key, hash)
292     }
293   }
294 }
```

Figure 4.6: Code for lock-free hash trie `get`.

The linearization point of `clone` is a successful CAS of `rootRef` at Line 262. In fact, the linearization point of all mutating *`LockFreeHashTrie`* methods will include `rootRef` in a CAS or RDCSS.

4.2.3 Hash trie `get(`k`)`

Figure 4.6 shows code for walking the trie to locate the value associated with a key. `get` uses Scala's *`Option`* type to indicate whether or not the key is present in the map, returning `Some(`v`)` if the value v is found and returning `None` if the key is not present.

Line 284 checks that `rootRef` hasn't changed since the original read (Line 275). If the root has been changed then there must have been a concurrent `clone`, in which case a tail recursive call is made to retry the search (Line 285).

The linearization point of `get` is the first *`Ref`*`.get` that returns a frozen *`Node`*, either on Line 275 or 282. Once the search's current position is a *`Leaf`* (always frozen) or frozen

Branch then the outcome is certain. Because references to an unfrozen branch cannot change except during `clone`, it is guaranteed that delaying all of the previous *Ref* reads to the linearization point does not affect the execution.

The observant reader will note that `get` would still be linearizable even if it ignored concurrent calls to `clone`. If any of the `Ref` reads were performed on a frozen *Branch* we could define the linearization point to be the first `clone` to occur after the last non-frozen `Ref` access, without requiring that the search be restarted. The linearization point for `get` would then be the CAS performed by the concurrent cloning thread. We will see in Section 4.3, however, that we would not be able to correctly execute the resulting algorithm simultaneously inside and outside a transaction. The version of `get` presented here always linearizes on an access performed by the current thread, which will allow us to compose the algorithm's operation by running them in an STM.

4.2.4 Hash trie put(k, v)

Figure 4.7 shows the trie code for inserting or updating a key–value association. Care is taken to avoid calling `unfreeze` on the root. Leaves are split into a branch inside *Leaf*`.withPut`, which can return either a *Leaf* or a *Branch* depending on whether the leaf capacity *LC* has been exceeded. `put` is written in a tail-recursive style. If it were written in a more imperative style there would be two loops, one for walking down the tree (corresponding to the recursion on Line 317) and one for retrying from the root (Line 312).

An interesting feature of this code is that the return value from the CAS on Line 328 is ignored. There is only one update possible for a *Ref* that points to a frozen *Branch*, which is to make a copy that homogenizes the generations of the parent and the child. The only way that the CAS can fail is if a concurrent thread successfully performs this transition, so regardless of success or failure the following *Ref*`.get` will obtain the *Branch* with the correct generation.

Any writes performed by `unfreeze` do not affect the abstract state of the trie. It is guaranteed that the branch that is coped on Line 328 is frozen; the root generation increases monotonically, so once a branch's generation doesn't equal the root generation it

```
295 class LockFreeHashTrie[A, B] ... {
296
297   def put(key: A, value: B): Option[B] =
298       put(rootRef.get, rootRef, 0, key, key.##, value)
299
300   @tailrec private def put(root: Node[A, B],
301                            nodeRef: Ref[Node[A, B]],
302                            shift: Int,
303                            key: A,
304                            hash: Int,
305                            value: B): Option[B] = {
306     (if (shift == 0) root else nodeRef.get) match {
307       case leaf: Leaf[A, B] => {
308         val after = leaf.withPut(root.gen, key, hash, value)
309         if (leaf == after || Ref.rdcss(rootRef, root, nodeRef, leaf, after))
310           leaf.get(key, hash) // no change or successful RDCSS
311         else
312           put(rootRef.get, rootRef, 0, key, hash, value) // retry from root
313       }
314       case branch: Branch[A, B] => {
315         val b = unfreeze(root.gen, nodeRef, branch)
316         put(root, b.childRefs(chIndex(shift, hash)), shift + LogBF,
317             key, hash, value)
318       }
319     }
320   }
321
322   private def unfreeze(rootGen: Long,
323                        nodeRef: Ref[Node[A, B]],
324                        branch: Branch[A, B]): Branch[A, B] = {
325     if (branch.gen == rootGen)
326       branch
327     else {
328       nodeRef.cas(branch, branch.copy(rootGen))
329       nodeRef.get.asInstanceOf[Branch[A, B]]
330     }
331   }
332 }
```

Figure 4.7: Code for lock-free hash trie put.

can never again equal the root generation. It is possible that `unfreeze`'s CAS is an update to a *`Branch`* that is itself frozen (because `rootRef` is not rechecked until a *`Leaf`* is encountered), but instance identity is irrelevant for frozen nodes so this is okay. It is only important that parallel contexts have a consensus on mutable instances. This is guaranteed because the CAS on Line 328 can only succeed once.

`put` linearizes at a successful RDCSS on Line 309. The extra compare in this operation is used to check that there has not been a call to `clone` since the traversal from the root was begun. If there has been a concurrent `clone` then Line 312 performs a tail recursive call that causes a retry from the root.

4.2.5 Hash trie `remove(`k`)`

The implementation of *`LockFreeHashTrie`*`.remove` follows the same outline as `put`, so we omit the code[3]. In the current implementation branches are never removed, even if all of their leaves become empty. This is less of a concern than it would be in an ordered tree because the uniform distribution of hash codes allows branches to be reused even when keys are always unique. Waste due to unused branches will only occur if the trie is large and then becomes permanently small.

Our ultimate execution environment for this algorithm is a system with TM, so we could fall back to a transaction to perform an atomic merging of a branch's children. The difficulty lies in tracking a branch's occupancy without reducing performance or introducing false conflicts. One possibility would be to opportunistically perform collapsing during copy-on-write. Extra snapshots could be triggered automatically by tracking trie statistics in a probabilistic or racy manner. We leave this for future work, and note that because leaves hold multiple elements the wasted space is only a small fraction of the nodes of a true tree.

[3] All of the code developed in this thesis is available under a BSD license. The transactional hash trie code is part of ScalaSTM's `github` repository [13].

4.2.6 Hash trie `size`

Implementing an efficient linearizable `size` operation is challenging for concurrent collections. *`ConcurrentSkipListMap`*, for example, falls back on an $O(n)$ algorithm that traverses all of its elements. The resulting `size` is not even linearizable, because the traversal is not consistent.

We implement `size` on the hash trie by taking a snapshot of the trie and then visiting all of the leaves. Because the snapshot is frozen the result is linearizable. This computation is still $O(n)$, but because *`Leaf`*`.size` is simply a lookup the constant factor is better than for a skip list. If there are multiple calls to `size`, however, we can do better. We can cache the size of each subtree in frozen *`Branch`* instances. If the next call to `size` is made after w updates to the trie then only those branches that have been unfrozen must be rechecked, so `size`'s running time will be $O(\min(w,n))$.

4.3 Lock-free Algorithms Inside Transactions

One of the original motivations for STM was to mechanically generate lock-free implementations of arbitrary data structures where no manually constructed lock-free algorithms were known, such as Herlihy et al.'s early example of binary search trees [44]. Composability is an added benefit of the resulting construction.

To provide composability for the lock-free trie of Section 4.2 we will perform a related transformation using TM, but instead of starting with a sequential algorithm we will start with the non-composable concurrent one. Our TM integration will enable hash trie methods to be called either inside or outside transactions. This will allow composition of the hash trie's linearizable operations when desired, while avoiding the overheads of TM when methods are called individually.

4.3.1 Semantics of non-transactional memory accesses

We assume that the TM is strongly atomic, also known as strongly isolated. Strong atomicity is included in all hardware TM proposals of which the author is aware. For software TMs strong atomicity can be provided by instrumenting all non-transactional accesses, but

the overhead is prohibitively high [49]. There have been several proposals for reducing this performance penalty using static analysis [83] or dynamic analysis [17, 80], but since `LockFreeHashTrie` has isolated all shared mutable state inside `Ref` instances we can rely on the type system as in Haskell's STM [39], CCSTM [16] or ScalaSTM [13].

Most lock-free algorithms require a means to atomically perform a read and a write, either the LL/SC (load-linked and store-conditional) pair or CAS (compare-and-swap). These can be implemented using a small atomic block, but it is not difficult to extend an STM to natively support something similar. ScalaSTM provides `compareAndSet`, which provides slightly less information than CAS but is sufficient.

4.3.2 Forward progress

Intuitively, transactions are just a limitation on the possible thread schedules. The essence of lock-free algorithms is that they are robust to the system's scheduling decisions, so there should be no problems mixing transactional and non-transactional invocations of a linearizable lock-free object's methods.

The intuitive forward progress argument is correct if memory accesses are individual steps in the execution. Moore et al. formalized this with a small-step operational semantics that limits state transitions when a transaction is active [66]. If the non-transactional memory accesses are implemented by an STM's isolation barriers, however, multiple steps are actually required to implement an individual memory access. Many recent STMs are not even obstruction free, as it appears that the practical overheads of STM can be lower in a blocking implementation [27]. We leave a formal treatment of this complication for future work. We note that it is similar in spirit to the accepted practice of coding lock-free algorithms for a virtual machine such as the HotSpot JVM whose garbage collector is not lock free, or of analyzing algorithms by treating all memory accesses as single steps, despite the seven orders of magnitude difference between the latency of an L1 cache hit and a hard page fault[4].

[4]The author cautions the gentle reader not to calculate the latency that results when a full garbage collection triggers hard page faults in random order for every rarely-used page of a program's heap.

```
333 class LockFreeHashTrie[A, B] ... {
334
335   @tailrec private def externalLPGet(nodeRef: Ref[Node[A, B]],
336                                      shift: Int,
337                                      key: A,
338                                      hash: Int): Option[B] = {
339     nodeRef.get match {
340       case leaf: Leaf[A, B] => leaf.get(key, hash)
341       case branch: Branch[A, B] => externalLPGet(
342           branch.childRefs(chIndex(shift, hash)), shift + LogBF,
343           key, hash)
344     }
345   }
346 }
```

Figure 4.8: Code for lock-free hash trie `get` whose linearization point may be inside a concurrent thread's `clone`.

4.3.3 Correctness after composition

Linearizability is often proved by defining the linearization points of each operation, the moment at which the operation appears to have executed atomically. It is not necessary that the linearization point be an instruction executed by the thread that called the operation, it is only required that the linearization point occurs after the operation was begun and before it is completed. Lock-free algorithms often have these external linearization points. They can arise when helping is used to implement compound mutations, or when a reading operation finds a stale value that must have been fresh at some time after the operation was begun.

Executing a series of linearizable operations inside a transaction is not sufficient to guarantee that they will appear to occur atomically. Let a_i and b be invocations of a linearizable object's operations, where $a_1, a_2 \ldots a_n$ are performed inside a single transaction by thread T and b is performed outside a transaction by thread U. If b is begun before the transaction starts and completed after the transaction ends, then linearizability allows b's linearization point to be a memory access performed by an a_j. In that case b will have violated the atomicity (if b is a read) or isolation (if b is a write) of the transaction.

To demonstrate this, we will describe a specific example of this problem that could occur if the hash trie's `get` did not check for concurrent calls to `clone`. Figure 4.8 shows the code for this version. In Section 4.2.3 we defined the linearization point as the first *`Ref`*`.get` to return a frozen *`Node`*, relying on the absence of a concurrent `clone` to avoid

the possibility that a node could become frozen after it had been located but before the search had traversed to its child. We can define `externalLPGet`'s linearization point to be the first memory access (by any thread) that causes the searches current position to be a frozen *Node*. This access can be either a *Ref*`.get` that moves the position to a frozen *Node* or a concurrent `clone` that causes the current *Node* instance to become frozen.

The code in Figure 4.8 is simpler, faster and has better progress guarantees (wait freedom) than that from Figure 4.6. Except for the problems introduced in this section, it would be a better implementation of `get`.

Linearizing `get(`k`)` at a concurrent `clone` will be a problem if the value associated with k at the time of the clone should not otherwise be visible. This can occur if a transaction calls `put(`k, v_1`)` then `clone` then `put(`k, v_2`)`. If a concurrent call to `get(`k`)` observes v_1, then the transaction's atomicity has been violated.

This apparent violation of atomicity is in fact possible. Consider a trie that consists of a single *Branch* containing *Leaf* instances. If the non-transactional `externalLPGet` locates the branch before the transaction runs but does not read the contained *Leaf* reference until after the transaction, the branch will contain the updated leaf from `put(`k, v_1`)` but not the updated leaf from `put(`k, v_2`)`. The second `put` must copy the branch before updating it, because it was frozen.

4.3.4 Atomicity violation – what is to blame?

The atomicity violation occurs only at the level of linearizable operations, atomicity and isolation are still preserved for the individual loads and stores. Something has gone wrong, but who or what is to blame?

- Serializability – Perhaps the problem is that TM provides guarantees only at the structural level, while programs are free to assign meaning to inferred virtual events that have no structural analogue. In this view the problem is a fundamental limitation of automatic concurrency control.

- Linearizability – An alternative is to view the problem as a limitation on the composability of linearizability. Speculative lock elision allows transactional execution of lock-based concurrent algorithms to compose with non-transactional execution [74], so perhaps linearizability gives too much freedom to the implementer? This view is not likely to be very attractive, because implementation freedom leads to more performant and scalable solutions.

- Strong atomicity – Under single global lock atomicity (SGLA) we would expect a transaction executed by T to constrain U's non-transactional execution [63]. In this model the observed behavior is allowed, and an object's operations can only be composed using transactions if transactions are always used. This view would be more compelling if there was not a subset of lock-free algorithms that could be correctly composed under strong atomicity but not under SGLA.

- Proof composition – One way to resolve this problem is to allow linearizable operations to be composed in a transaction, but to consider the composition to be a new operation whose linearizability must be proved separately. Here the blame is placed not on linearizability or serializability, but on the expectation that the composable atomicity and isolation of memory transactions extends to composition of linearizability proofs. Of course proving linearizability of the resulting compound operations will be very difficult.

- Premature optimization – Perhaps the fault lies with the author of this thesis, for attempting to remove some transactions to get only a constant factor improvement in performance.

Regardless of the cause, it is important to characterize the problem so that we may tame or avoid it.

4.3.5 Internal linearization points – a sufficient condition

We say that an operation is *internally linearizable* if the linearization point (LP) can only be a memory access performed by the thread executing the operation[5]. Strong atomicity guarantees that a memory transaction is isolated from accesses by other threads, so a transaction cannot contain the LP of an internally linearizable operation performed by a concurrent thread. This is sufficient to guarantee that a non-transactional internally linearizable operation cannot appear to be interleaved with the transaction's operations.

Internal linearizability is not the most general property that guarantees that non-transactional operations cannot be linearized inside a concurrent transaction. A devil's advocate algorithm might try for a fixed number of instructions to locate and communicate with concurrent invocations, falling back to a lock-free internally linearizable implementation if no partner could be found. Operations on this byzantine object would not be able to conclusively detect that they were inside a transaction, but they would sometimes be able to prove that they were outside. They could then arrange that an external LP could only occur outside a transaction, preventing the transaction anomaly without being internally linearizable.

The transactional hash trie with snapshots always linearizes on a `Ref` access performed by the current thread, so it is internally linearizable. All of its methods may be invoked either inside or outside a transaction with the expected atomicity and isolation.

4.4 Transactional Access to the Hash Trie

Transactional integration of the lock-free hash trie is as simple as executing the algorithm inside a transaction when composition is required. All shared mutable state is encapsulated in `Ref` instances, making it easy to use a library-based STM.

[5]The LP can always be defined to be the completion of some memory access, because an LP at an instruction that does not affect the shared state of the machine is equivalent to an LP at the preceding instruction.

4.4.1 `clone` inside a transaction

The hash trie's snapshot-based `clone` works fine when called from inside a transaction; the resulting clone acts as if the entire trie was copied but requires only $O(1)$ work. `clone` writes to the root to ensure that the shared content is marked frozen, however, which means that a transaction that calls `clone` will conflict with any other transaction that accesses the map. Multiple calls to `clone` will also conflict with each other, because each will attempt to install a new root to create a new snapshot.

Snapshots remain valid until a subsequent write, so we can improve the situation by advancing the root generation only if some of the children are not frozen. If we add a bit to *`Branch`* that allows the root branch to be considered frozen, then we can go even further and allow duplicate snapshots to reuse the entire trie. With this extension `clone` does not increment the generation number, but rather installs a frozen root *`Branch`*. Copy-on-write must then be performed on the root branch prior to the next write. This scheme has the advantage that if there are no mutating operations performed on the map then all map operations will become read-only.

4.4.2 Optimizations for transactional operations

While the lock-free code works correctly inside a transaction, the transaction's isolation allows the code to be simplified. There is no need for `get` to retest `rootRef` (Line 285), because it can't have changed after its original read. Similarly there is no need to use CAS or RDCSS in any operation, because a *`Ref`*`.get` followed by *`Ref`*`.set` will have the same effect.

Scala's collection classes are both iterable and traversable. Iterable collections can return an *`Iterator`* instance that will produce the elements of the collection one at a time. Traversable collections define a `foreach` method that will invoke a user-supplied method for each element. The hash trie's iterator must always take a snapshot because the iterator may escape the scope of a transaction. Transactional `foreach`, however, may skip the snapshot and rely entirely on the transaction's isolation.

4.4.3 Adaptively reducing false conflicts

The hash trie's packed leaves help to improve cache locality and reduce memory use, but they can introduce false conflicts. One of the compelling benefits of transactional predication (Chapter 3) was a reduction in false conflicts, and we would like transactional hash tries to be similarly robust.

False conflicts can be avoided by splitting contended leaves. Most STMs use invisible readers, which makes it difficult for writing transactions to detect that they are causing conflicts with readers. Maps that are experiencing read-write conflicts are likely to experience some write-write conflicts, which can be efficiently detected in several high performance STMs including McRT-STM [79] and SwissTM [25]. Our experimental evaluation is performed using ScalaSTM, which is based on the SwissTM algorithm.

ScalaSTM provides a weak form of `Ref.set` called `trySet`, which returns false and does nothing if another transaction already has acquired ownership of a *Ref*. This detects both write-write conflicts and read-write conflicts where the reader has fallen back to ScalaSTM's pessimistic read mode to guarantee forward progress.

We use failed `trySet` to detect false conflicts. The thread that detects the false conflict is not in a position to fix the problem, unfortunately. If it tries to split the leaf it is guaranteed to trigger a rollback, and if it rolls itself back there will be no improvement. Also, write-write conflicts are evidence that undetected read-write conflicts are likely. Rather than using the results of a failed `trySet` locally, we use it to maintain a per-map exponential moving average of the rate of write-write conflicts. Each call to `put` then uses a more aggressive splitting cutoff if this estimate exceeds a fixed threshold.

The parameters of the contention estimate are dependent on the details of the STM. For ScalaSTM reasonable behavior is observed on all of the microbenchmarks from Chapter 3 with an exponential moving average that retains $1 - \frac{1}{512}$ of its previous value during each `put` and that aggressively splits leaves whenever the average indicates more than 1% of calls to `trySet` fail.

Of course, even a racy update of a global contention estimate during each call to `put` would limit the scalability of the trie. Because `trySet` failures are rare we wish to count them exactly, but successes are common so we can update the estimate probablistically.

```
347 class LockFreeHashTrie[A, B] ... {
348   private def pct = 10000
349   private def conflictThreshold = 1 * pct
350
351   // Ranges from 0 (no conflicts) to 100 * pct (conflict every time)
352   private var conflictEstimate = 0
353
354   private def recordNoconflict() {
355     if (ThreadLocalRandom.nextInt(1 << 5) == 0) {
356       val e = conflictEstimate
357       conflictEstimate = e - (e >> 4)
358     }
359   }
360
361   private def recordconflict() {
362     val e = conflictEstimate
363     conflictEstimate = e + ((100 * pct - e) >> 9)
364   }
365
366   private def adaptiveLeafCapacity: Int = {
367     if (conflictEstimate > conflictThreshold) 1 else LC
368   }
369
370   private def txnSet(ref: Ref[Node[A, B]], node: Node[A, B]) {
371     if (!ref.trySet(node)) {
372       recordconflict()
373       ref.set(node)
374     } else
375       recordNoconflict()
376   }
377 }
```

Figure 4.9: Code to track the rate of write-write conflicts using a probabilistically-updated exponential moving average with $\alpha = 2^{-9}$.

We use a thread-local random number generator to record 32 successes $\frac{1}{32}$ of the time. Figure 4.9 shows the code used inside a transaction to update a `Leaf`. Fractional values are stored out of 1,000,000 to allow us to use integer arithmetic. We use the approximation that $1-(1-\frac{1}{512})^{32} \approx \frac{32}{512} = \frac{1}{16}$, which is correct within a few percent.

4.5 *TMap* Recipes

In this section we will illustrate the composability features of the transactional map by giving recipes for useful compound operations. These examples use ScalaSTM's API,

which includes our transactional hash trie as the default implementation of its *TMap*[***A***,***B***] interface [13].

4.5.1 ScalaSTM's types

The two most useful container types in ScalaSTM are *Ref*[***A***] and *TMap*[***A***,***B***]. *Ref* is a mutable transactional reference that holds an instance of ***A***. *TMap* is a transactional map from instances of ***A*** to ***B***, where each instance of ***A*** passed to the map must not be changed. ScalaSTM's *TMap* factory instance returns transactional hash tries that use the algorithm introduced in this chapter.

ScalaSTM uses the type system to statically check that transactional operations are only called inside an atomic block, as in Haskell's STM [39] and CCSTM [16]. This is accomplished by requiring an implicit *InTxn* parameter on each transactional method. The Scala language automatically connects implicit declarations to implicit parameters, so the implicit *InTxn* instance passed into the atomic block need not explicitly appear at the call site. The following three snippets are translated identically by the Scala compiler:

```
val m: TMap[Int, String] = ...

// the InTxn witness is passed explicitly
atomic { (t: InTxn) =>
  if (m.contains(0)(t))
    m.put(10, "ten")(t)
}

// the InTxn witness is passed explicitly, with type inference
atomic { t =>
  if (m.contains(0)(t))
    m.put(10, "ten")(t)
}

// the InTxn witness is passed implicitly
atomic { implicit t =>
  if (m.contains(0))
    m.put(10, "ten")
}
```

When only a single operation would be performed by an atomic block, ScalaSTM provides an alternate set of types whose methods do not require an implicit *InTxn* parameter. Access to this type is obtained via a `.single` method on the transactional instances. *Ref*'s `single` returns an instance of *Ref.View*, and *TMap*'s `single` returns an instance of type *TMap.View*, which is integrated into Scala's mutable collection hierarchy as a subclass of *scala.collection.mutable.Map*. The returned type still supports optional composition using transactions, with the surrounding transaction scope located by a dynamic check.

4.5.2 *TMap* as a normal concurrent map

To construct a transactional map from integers to strings, that can be used either inside or outside a transaction. A dynamic search will be made at each call (using a *ThreadLocal*) for an enclosing atomic block:

```
val m = TMap.empty[Int, String].single
```

The returned instance `m` may be used directly as a concurrent map, with no atomic blocks:

```
val existing = m(key)          // access an existing element
val maybe = m.get(key)         // Some(v) if present, None otherwise
m += (k -> v)                  // add or update an association
val maybePrev = m.put(k, v)    // Some(prev) on update, None on insert
```

Some methods of Scala's mutable *Map* perform multiple accesses, such as `getOrElseUpdate` and `transform`. *TMap.View*'s implementation of these is atomic, using transactions underneath as necessary.

4.5.3 Consistent iteration and immutable snapshots

Most concurrent data structures (including all of those in *java.util.concurrent*) do not provide consistent iteration. `TMap.View`, however, always takes a snapshot prior to iteration. This means that no extra work is required, and all of the powerful functional features of Scala's maps inherit the increased safety of snapshot isolation:

```
for ((k, v) <- m) { ... }          // snapshot isolation
val atomicSum = m.reduceLeft(_+_)  // reduction is linearizable
```

If an explicit snapshot is required, `m.snapshot` uses the copy-on-write mechanism to return an instance of *`scala.collection.immutable.Map`*.

4.5.4 Compound operations on a single map

Concurrent maps often support atomic modification of the value associated with a key with a CAS-like operation. If a map's declared value type is ***B***, then the actual range of existing and new values that may be passed to CAS is ***B***+1, where the extra inhabitant encodes the absence of an association. In Scala this type is referred to as *`Option`*[***B***]. Java's *`ConcurrentHashMap`* encodes the extra inhabitant of the value type by separating the functionality of CAS over four methods:

expected	*new*	CAS(k, *expected*, *new*)
None	None	!containsKey(k)
None	Some(v_1)	putIfAbsent(k, v_1) == null
Some(v_0)	Some(v_1)	replace(k, v_0, v_1)
Some(v_0)	None	remove(k, v_0)

The CAS-like operations can be combined with an optimistic retry loop to code arbitrary transformations of an immutable value, but they can't be used to modify the key. The transactional map makes this trivial. In the following code `remove` returns an *`Option`*[***B***], and the **for** comprehension over the *`Option`* traverses either 0 or 1 value. This makes sure that `k1`'s association is only updated if `k0` is present:

```
def renameKey(k0: A, k1: A) {
  atomic { implicit txn =>
    for (v <- m.remove(k0)) m(k1) = v
  }
}
```

Scala's mutable *`Map`* interface provides some methods that perform bulk mutation of the map, including `++=`, `transform` and `retain`. *`TMap.View`* provides atomicity and isolation at the method level for all of *`Map`*'s methods, so all of these bulk methods execute without visible concurrency interleavings:

```
m.retain { (k,v) => v.isValid }  // atomically remove invalid entries
```

4.5.5 Compound operations across multiple maps

Composability inside an atomic block extends across all STM-managed data, including multiple *Ref*s, *Ref.View*s, *TMap*s, and *TMap.View*s. This makes it easy to perform multi-step operations like moving elements between maps, or checking that a map is empty and then removing a reference to it:

```
// move entries from one generation to another, without any
// entry being visible in two generations
atomic { implicit txn =>
  oldGen ++= eden
  eden.clear
}
```

4.5.6 Synchronized forward and reverse indices

Maps are suitable for both primary data storage and for implementing indices. When used as a primary store, the map key is the primary identifier of the data and the value holds a single record. When used as an index, the map associates the indexed value with a set of records.

As a simple example, consider a phone system that uses the phone number as the primary key, and that also has an index that allows looking up numbers by name. We'll let each phone number have multiple associated names. In Scala we can code an immutable record concisely using a **case class**, which acts like a tuple with named fields:

```
type Number = String
type Name = String
case class Record(number: Number, names: immutable.Set[Name], ...)
```

The primary storage for records is a map from number to *Record*:

```
val records = TMap.empty[Number, Record]
```

The index by name is not unique, because each name might be associated with more than one phone. Because we are operating in memory, the index can link directly to the target records without any wasted space:

```
val byName = TMap.empty[Name, immutable.Set[Record]]
```

Inserting

To insert a *Record* in a thread-safe fashion, we must atomically insert an entry into the `records` map and insert or update the affected entries in the `byName` map. The hash trie's transactional integration makes this as easy as the equivalent sequential code:

```
def insert(rec: Record) {
  atomic { implicit txn =>
    records(rec.number) = rec
    for (n <- rec.names)
      byName(n) = byName.getOrElse(n, immutable.Set.empty[Record]) + rec
  }
}
```

Removing

Removing a *Record* also requires composing a modification of `records` with modifications of `byName`. Notice that because the atomic block is isolated from outside interactions, we can use sequential reasoning to detect when the last record is removed from the index for a name:

```
def remove(rec: Record) {
  atomic { implicit txn =>
    records -= rec.number
    for (n <- rec.names) {
      val prev = byName(n)
      if (prev.size == 1)
        byName -= n
      else
        byName(n) = prev - rec
    }
  }
}
```

Updates

Updating a record is equivalent to atomically removing and then inserting it. We can simplify this interface for the caller if we fetch the previous value (the one to be removed) using

the new record's `number`. This also allows us to combine the functionality of insertion and update into a single `put` method, as is common in map implementations. Atomic blocks nest, so we can reuse `insert` and `remove`:

```
def put(rec: Record) {
  atomic { implicit txn =>
    for (prev <- records.get(rec.number))
      remove(prev)
    insert(rec)
  }
}
```

Lookup

Queries by a single name or by phone number are likely to constitute the vast majority of the operations on our phone number storage. Because the values stored in the transactional maps are immutable, and only a single map needs to be accessed to perform a lookup, no atomic block is required:

```
// returns None if not found
def lookupByNumber(num: Number): Option[Record] =
    records.single.get(num)

// returns an empty Set if not found
def lookupByName(name: Name): Set[Record] =
    byName.single.getOrElse(name, Set.empty[Record])
```

Snapshots

Suppose that we would like to capture the current state of the phone directory, perhaps to render an HTML by-name and by-number listing in a web interface. We can guarantee that the two portions of the result will be in sync by atomically capturing a snapshot of both transactional hash tries:

```
def snapshot = atomic { implicit txn => (records.snapshot, byName.snapshot) }
```

Scala will infer a return type for this method of:

```
(immutable.Map[Number, Record], immutable.Map[Name, immutable.Set[Record]])
```

4.6 Experimental Evaluation

In this section we evaluate the performance of the transactional hash trie. First we reproduce the throughput micro-benchmarks from Chapter 3, finding that the transactional hash trie's average performance is close to that of the predicated `ConcurrentHashMap` despite the hash trie's extra features. Second, we use the synthetic STMBench7 benchmark to compare transactional use of the hash trie to sophisticated non-composable pessimistic locking. Finally, we examine the use of transactional hash tries and STM in a large real-world application by rewriting Apache Derby's lock manager. Our code using memory transactions is simpler, avoids subtle correctness arguments, scales, and achieves 98% of the peak performance of the original fine-grained locking version.

Experiments were run on a Dell Precision T7500n with two quad-core 2.66Ghz Intel Xeon X5550 processors, and 24GB of RAM. We used the Linux kernel version `2.6.28-18-server`. Hyper-Threading was enabled, yielding a total of 16 hardware thread contexts. Scala code was compiled with Scala version 2.9.0-1. We ran our experiments in Sun's Java SE Runtime Environment 1.6.0_26, using the HotSpot 64-Bit Server VM with default JVM options (the JVM memory flags usually added by the `scala` launcher script were not used).

We used ScalaSTM 0.4-SNAPSHOT (git version `10535a48`) in its default configuration [13]. ScalaSTM's reference implementation was originally derived from CCSTM [16]. The STM uses the SwissTM algorithm [25], which is a variant of TL2 [24] that detects write-write conflicts eagerly.

4.6.1 Differences between ScalaSTM and CCSTM

While the version of ScalaSTM used in the experimental evaluation of this chapter is derived from the CCSTM used in Chapter 3, it includes an important improvement in its contention management handling. When atomic blocks in CCSTM have experienced several rollbacks due to inconsistent reads they enter a pessimistic read mode in which write locks are acquired for all memory accesses. Combined with transaction priorities, the pessimistic read mode guarantees that every transaction will eventually succeed. The implementation of pessimistic reads in CCSTM uses the write buffer, which causes a memory location's

version to be incremented on commit. This causes a cascading failure for tree-like structures, because pessimistic reads of the root trigger optimistic failures in concurrent readers, which will in turn result in more pessimistic readers. ScalaSTM uses a separate data structure to track pessimistic reads. This means that a transaction that is using the pessimistic read mode does not interfere with transactions that are still performing their reads optimistically.

ScalaSTM also differs from CCSTM by providing true nesting with partial rollback, which is required by its support for Harris et al.'s `retry` and `orElse` primitives [39]. A novel high-water mark is used inside the write buffer to optimize for the case when all writes to a *`Ref`* occur in the same nesting level. Despite its support for partial rollback, the additional engineering effort that has gone into ScalaSTM, Scala 2.9.0-1 and JRE 1.6.0_26 mean that transactions are in general faster than in our previous experiments.

4.6.2 Microbenchmarks

The microbenchmarks emulate the methodology used by Herlihy et al. [43]. Each pass consists of each thread performing 10^6 randomly chosen operations on a shared map; a new map is used for each pass. To simulate a variety of workloads, two parameters are varied: the proportion of `get`, `put` and `remove` operations, and the range from which the keys are uniformly selected. A smaller fraction of `gets` and a smaller key range both increase contention. Because `put` and `remove` are equally likely in our tests, the map size converges to half the key range. To allow for HotSpot's dynamic compilation, each experiment consists of twenty passes; the first ten warm up the VM and the second ten are timed. Throughput results are reported as operations per microsecond. Each experiment is run five times and the arithmetic average is reported as the final result.

Figure 4.10 replicates all of the experiments from Figure 3.5, adding results for 'snap-trie', the transactional hash trie with snapshots. All of the map implementations evaluated in Section 3.6 are included, updated to use ScalaSTM and Scala 2.9.0-1.

- **conc-hash** – Lea's *`ConcurrentHashMap`* [59], as included in the JRE's standard library;
- **boosting-soft** – a transactionally boosted *`ConcurrentHashMap`*. Soft references are used to reclaim the locks.

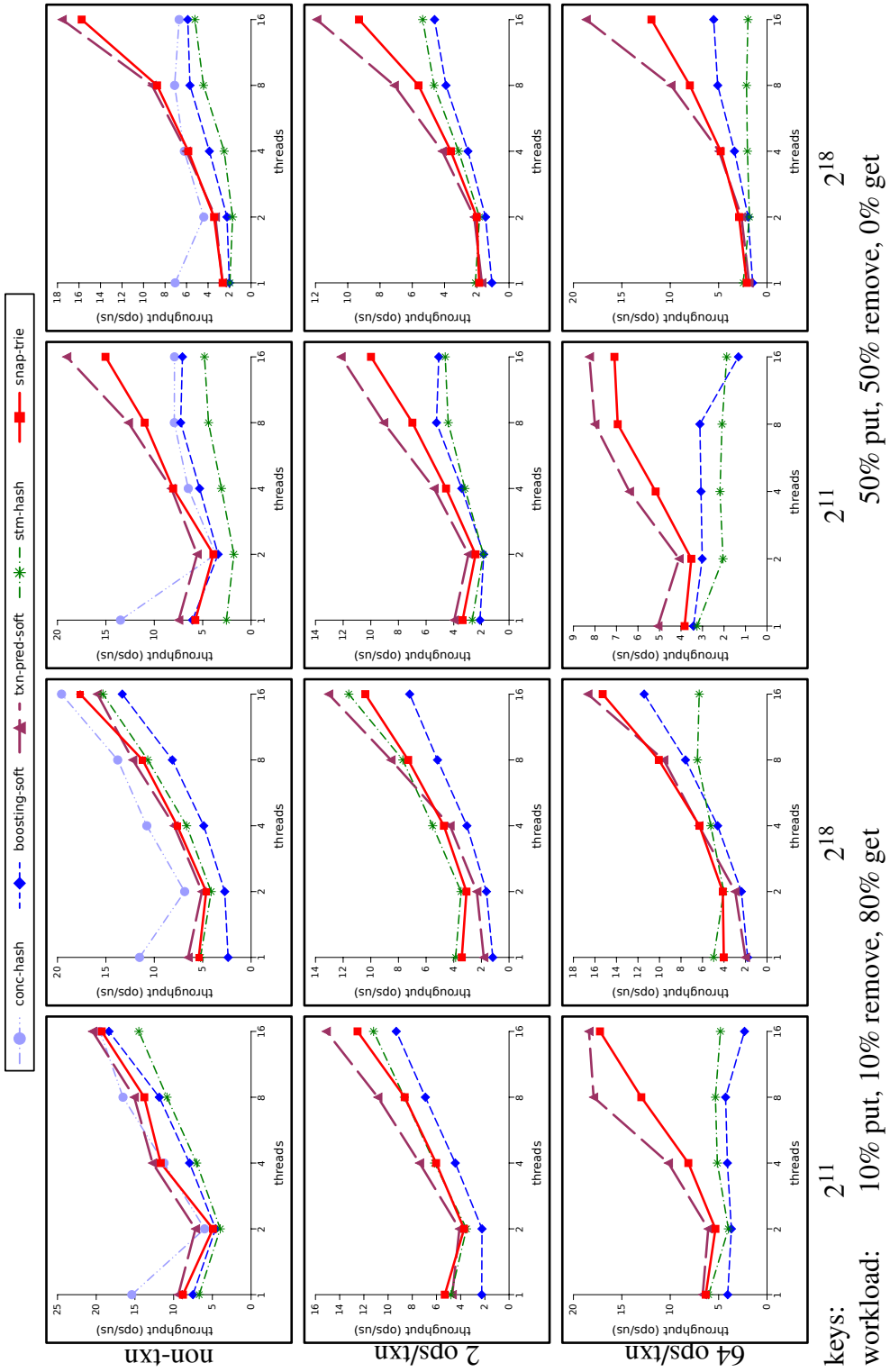


Figure 4.10: Throughput of transactional map implementations across a range of configurations. Each graph plots operations per microsecond, for thread counts from 1 to 16.

- **txn-pred-soft** – a predicated `ConcurrentHashMap` that uses the the soft reference mechanism of Section 3.4.3.
- **stm-hash** – a hash map with 16 segments, each of which is a resizeable transactional hash table.
- **snap-trie** – a transactional hash trie with snapshots, from this chapter.

The general relationship between transactional predication and the other transactional alternatives remains the same, despite changes in the STM, Scala language, and JVM. Although few rankings have changed we note that the absolute performance is higher across the board, so the y axes are not all the same.

Snap-trie is on average 18% slower than txn-pred-soft across the small key range. A small key range is a best case scenario for the soft reference reclamation mechanism, because there is no memory pressure. The performance difference between the hash trie and predication is reduced for the large key range; snap-trie's excellent performance for low thread counts means that the geometric mean of its throughput is actually 0.1% higher than transactional predication's for the large key range, despite being lower for most data points. Snap-trie's advantage is largest for transactional read-heavy access at low thread counts, where 45% of the calls read a non-existent key (half of the reads and half of the removes). Txn-pred-soft must lazily recreate predicates to handle read misses that occur inside a transaction. The optimizations of Section 3.4.4 minimize this penalty for access outside a transaction.

The hash trie's performance almost completely dominates both the transactional hash table and transactional boosting, and also outperforms `ConcurrentHashMap` for contended configurations[6]. Conc-hash has very good single-threaded performance because the JIT can remove almost all concurrency control code from the hot path, but we don't expect this to be a typical scenario.

Snap-trie offers comparable performance and scalability to transactional predication, while providing extra functionality. The microbenchmarks did not reveal any performance pathologies. In addition, it does not require cleanup threads, has a much smaller memory

[6]The JDK7 release includes a `ConcurrentHashMap` implementation with better performance under write contention, but JIT bugs currently make JDK7 problematic.

footprint when it contains only a few elements, and supports a linearizable `size` operation without requiring extra work for all operations, so we conclude that snap-trie is the best unordered transactional map for general purpose use.

4.6.3 STMBench7

STMBench7 is a synthetic STM benchmark that performs a mix of reads, writes, bulk reads, and bulk writes over an in-memory representation of a CAD model [34, 26]. Versions are available for C++ and for Java.

This benchmark is especially well suited for measuring the overheads of generic composability because it includes reference implementations that use hand-coded non-composable pessimistic locking. Two locking strategies are included: coarse and medium. The medium-grained locking implementation is representative of the approach that an experienced parallel programmer might take. It is complicated enough that early versions of the benchmark had concurrency bugs, despite being created by parallel programming researchers.

A majority of the transactional work performed by STMBench7 involves the creation, reading and updating of unordered transactional sets and maps. Since ScalaSTM doesn't perform bytecode rewriting, we implemented a straightforward STMBench7 adapter, in 365 lines of Scala. All unordered sets and maps were implemented using the transactional hash trie introduced in this chapter, as included in the ScalaSTM distribution. We used the Java STMBench7 version 1.2 (25.02.2011), disabled long traversals, and did not count operations in which *`OperationFailedException`* was thrown (this is the configuration used previously by STM researchers). Each data point is the average of 5 executions of the benchmark, each of which lasted for 60 seconds.

Performance

Let's start with the read-dominated workload, which should benefit most from the STM's optimistic locking. The test machine has 8 real cores plus Hyper-Threading, so it can run 16 threads at once. We also tested at higher thread counts to verify the robustness of the contention management in *`scala-stm`* and the transactional hash trie, and because

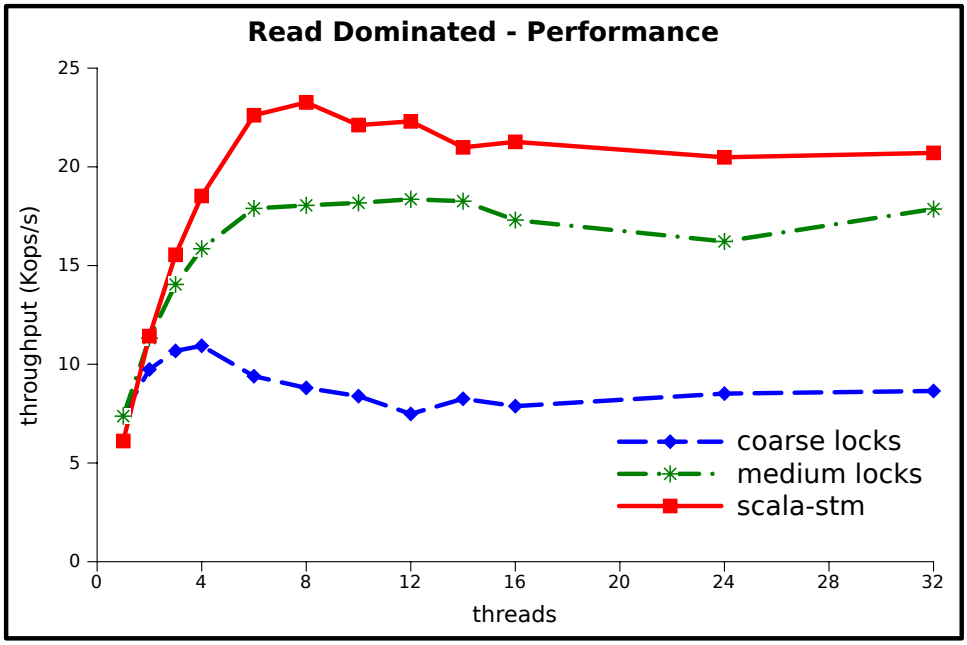


Figure 4.11: STMBench7 performance with a read dominated workload.

pessimistic locking strategies may achieve their peak performance with more threads than cores (if cores would otherwise be idle during blocking).

Figure 4.11 shows that for the read dominated workload, our STM implementation has 16% higher peak performance than even the complex medium-grained locks. The single-thread overhead incurred by the STM's automatic concurrency control and the extra features of the transactional hash trie (including composability and support for snapshots and consistent iteration) is overcome by the superior scalability of optimistic concurrency control. This is an excellent result, since the STM implementation is as easy to use for the programmer as the coarse-grained lock. Despite being much more complicated (which means expensive to write, test, and maintain) the medium-grained locks have a lower peak performance than the simple STM code.

We would not expect optimistic concurrency control to scale as well in a write-dominated workload with high contention. Figure 4.12 shows that none of the implementations is able to find much scalability. There is too much writing for readers to get a benefit from optimistic concurrency or reader/writer locks, and the writers touch the same data so often that any benefits from parallelism are lost to cache misses and synchronization overheads.

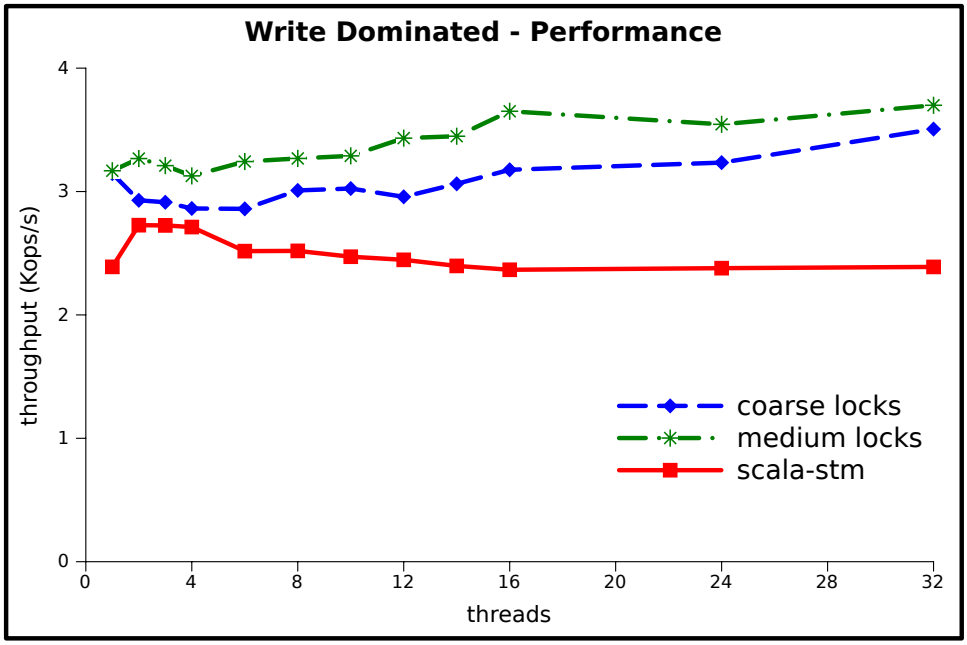


Figure 4.12: STMBench7 performance with a write dominated workload.

The STM can't use scalability to compensate for its single-thread overheads, but at least it is handling the contention without any performance pathologies. The STM's peak performance for the write-dominated workload is 31% worse than the medium-grained locks.

Throughput results for a mixed workload are shown in Figure 4.13. At first glance this looks more like the write-dominated performance than the read-dominated one, for all of the implementations. In fact, this result is as expected. We are seeing the read-world effect of Amdahl's law. We can think of the write-dominated component as the sequential part, since it doesn't benefit from parallelism, and the read-dominated component as the parallel part, since it gets faster when we add threads. 50% of the work isn't scalable, so we will be limited to a speedup of 2. Even if reads were completely free, the mixed workload's *ops*/*sec* would only be double the write-dominated workload's *ops*/*sec*.

We can compute an expected *sec*/*op* of the mix by average the *sec*/*op* of the two component workloads. Armed with this formula we see that the mixed workload's performance is exactly as would be expected from the previous two experiments.

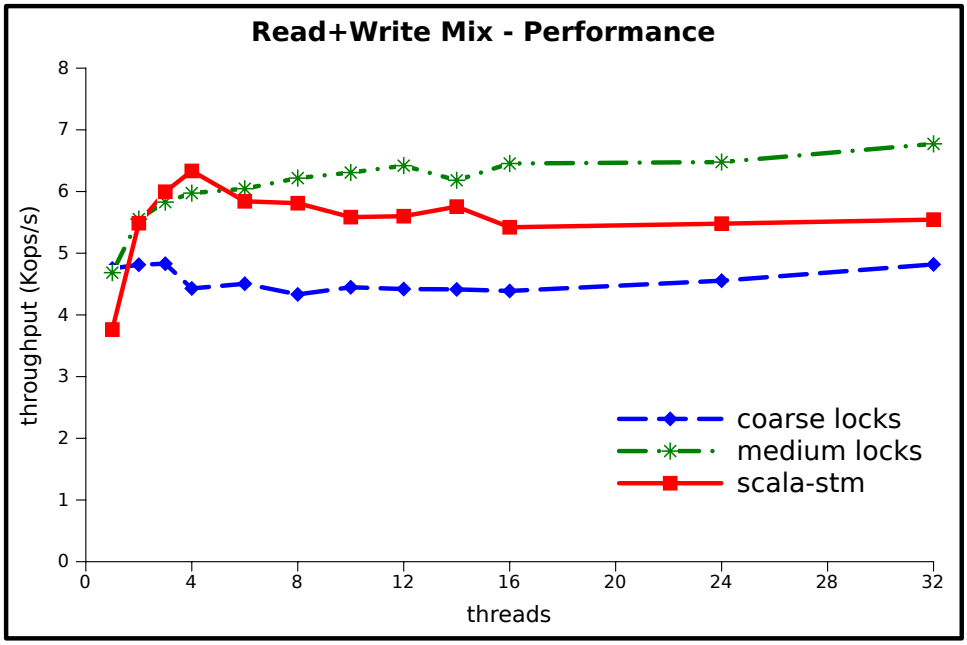


Figure 4.13: STMBench7 performance with a workload that is an even mix of those from Figures 4.11 and 4.12.

Garbage collector load

There are several potential sources of extra GC load in the `scala-stm` version of STMBench7:

- allocation from the closures that help make the Scala code aesthetic;
- allocation in the STM itself (although for small- and medium-sized transactions the ScalaSTM reference implementation allocates only one object per transaction);
- objects discarded when a transaction is rolled back and retried;
- short-lived wrapper instances needed to use the generic *`Ref`* interface on the JVM (these may eventually be targeted with Scala's `@specialized` annotation); and
- copying performed during updates to the *`immutable.TreeSet`*, *`TSet`* and *`TMap`* used by the Scala version of the benchmark.

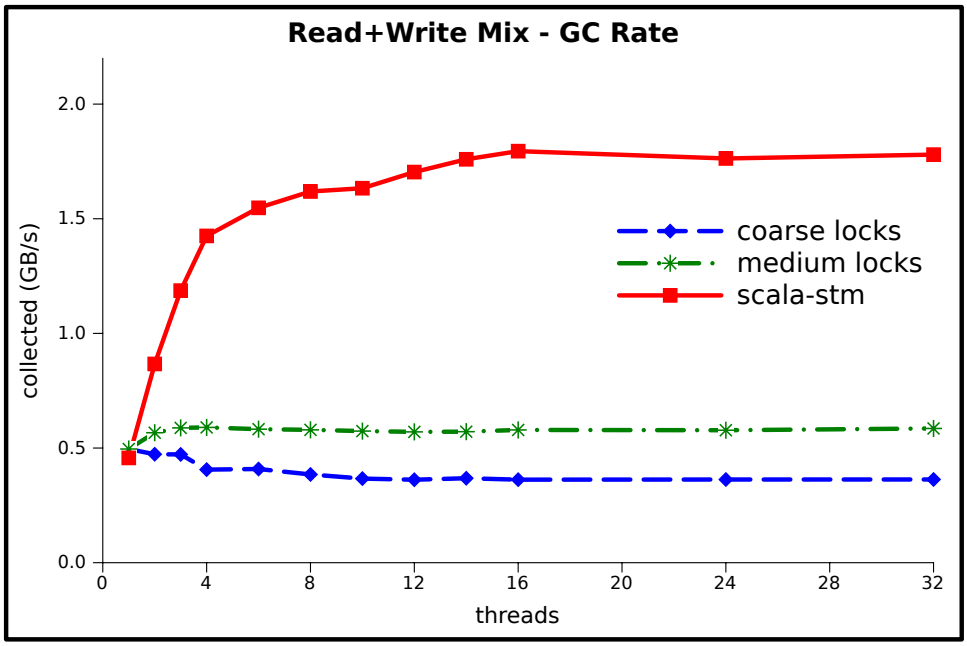


Figure 4.14: GC reclamation rate during STMBench7 for the mixed workload from Figure 4.13.

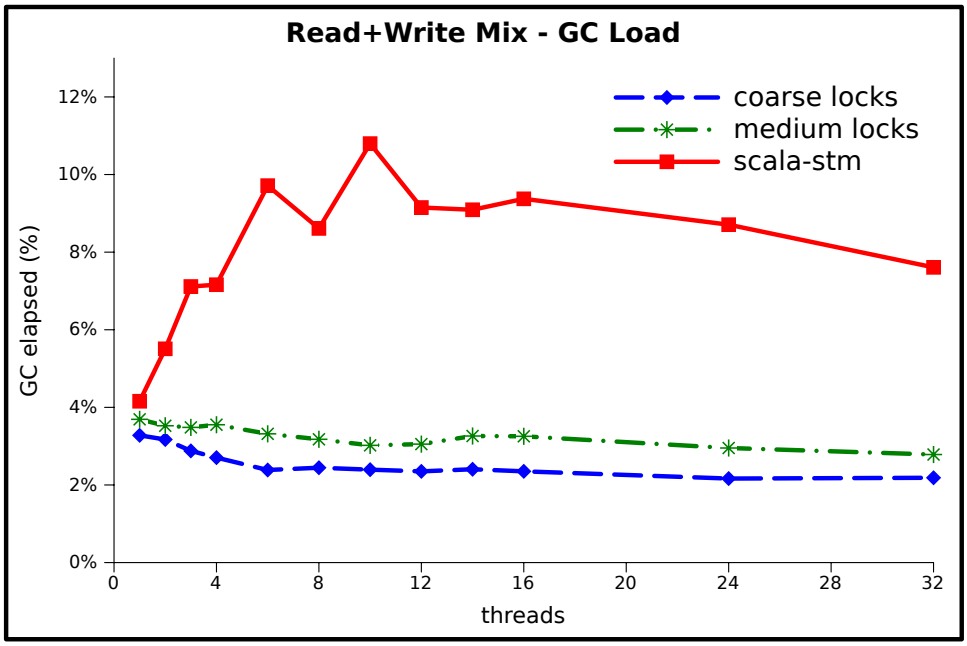


Figure 4.15: GC wall time, as a fraction of the total wall time for the mixed STMBench7 workload from Figure 4.13.

Figure 4.14 shows the steady-state reclamation performed by the GC during the benchmark, in gigabytes per second. The peak GC throughput is 3.2 times higher for STM than for the medium-grained lock implementation. Figure 4.15 shows a similar (2.9 $\times$) relative increase in the wall time used by the garbage collector. The results from other workloads are similar, with a peak GC load less than 11% for all cases. The benchmark represents a worst-case scenario in which all threads spend all of their time in transactions.

STMBench7 conclusion

This set of experiments shows that taking advantage of the transactional hash trie's composability can yield good performance and scalability even when the baseline is hand-coded pessimistic locking by parallelism experts (or at least parallelism researchers), and even when the underlying STM does not have the advantage of language, compiler, or VM integration. The carefully engineered lock-based implementation yielded higher performance for contended scenarios, but had less scalability and was substantially more complex. The extra GC load of the transactional hash trie could be reduced by an STM that was not implemented as a pure library. Since our experiments used a stop-the-world garbage collector, any time recovered from GC overhead would contribute directly to benchmark throughput; in an execution in which the GC overhead was reduced from 8% to 4% we would expect a $\frac{1-0.04}{1-0.08} - 1 \approx 4\%$ improvement in the STMBench7 score.

4.6.4 In-situ evaluation inside Apache Derby

The promise of memory transactions is good performance and scalability with a simpler programming model. Our work on transactional maps is an effort to deliver that promise by optimizing the data structure libraries that fit between the program and the STM. To fully demonstrate the success of this approach we must tackle a problem at scale.

Derby's lock manager

The lock manager inside Apache's Derby SQL database is a complicated concurrent component that is critical to the performance and scalability of the database. It is responsible for: granting row-level locks to database transactions; tracking the locks that must be

released after a database transaction is completed; managing named subgroups of locks; detecting and breaking deadlock cycles; fair queuing of lock waiters with opportunistic piggybacking; and timeouts.

The original implementation of Derby's `LockFactory` interface used coarse-grained Java monitors. It consisted of 2,102 non-comment lines of Java. Database lock acquisition and release was serialized, and all lock activities were blocked during deadlock detection. In 2007 this code was identified as a scaling problem (issue DERBY-2327) and it was replaced with a more complicated implementation that scales better. The replacement uses fine-grained locks and `ConcurrentHashMap`; it is only 102 lines longer than the coarse-locked version, but its logic is much more subtle. Ironically, 128 lines of mailing list discussion were devoted to informally proving that the deadlock detector is not itself subject to deadlock! The switch to fine-grained locks was successful; a test client designed to stress scalability shows bad scaling before the rewrite and excellent scaling afterward, with a negligible difference in single-thread performance.

The set of currently granted locks can be described as a multiset of 4-tuples $\langle r,t,g,q\rangle$, where r is a row, t is a transaction, g is a thread-local group of rows within a transaction and q is the lock type, such as shared or exclusive. The lock manager is generalized to support locking other types of objects as well, but the heaviest use is of these instances. The generic type that includes rows is `Lockable`. Database transactions are generalized as members of a `CompatabilitySpace`, and the lock type is referred to as a qualifier. Each `Lockable` r is also associated with a fair queue of waiting tuples. Individual acquires and releases are specified using the entire 4-tuple. Bulk release and existential queries are provided by t and $\langle t,g\rangle$.

Both the coarse- and fine-grained lock manager implementations make extensive use of mutable state. Indices to support efficient lookup by r, t and g are constructed in an ad-hoc manner, and state transitions such as handing a lock to the next waiter require multiple steps and the participation of both threads. Extra logic is required to handle the interaction of concurrent multi-step transitions, but this code is rarely executed and was the cause of at least one production concurrency bug (DERBY-4711). The following is a list of the corner cases we found while trying to understand the code; there are almost certainly more:

- Lock availability is a function of both the lock status and the wait queue, because multiple steps are required to hand the lock to the next waiter;
- Multi-step lock hand off has a complex interaction with timeout of a lock request (this is what caused DERBY-4711);
- Multi-step lock hand off has a complex interaction with piggybacking of lock requests from the same transaction, both when piggybacking becomes newly possible and when it becomes newly impossible;
- Lock requests can time out after they have been cancelled by the deadlock detector but before the cancel has been noticed by the waiting thread; and
- Deadlock detection acquires locks in encounter order, which necessitates an additional level of locking to restore a global lock order.

Scala implementation using the transactional hash trie

We completely reimplemented the *LockFactory* interface in Scala using transactional maps and immutable data structures. The only mutable shared instances in our implementation were transactional hash tries, which have type *TMap*.

The Scala reimplementation of the lock manager totals 418 non-comment lines of code. 67 of those lines are immutable implementations of Java interfaces; the remaining 351 lines implement *LockFactory*'s 14 methods and all of its functionality, including deadlock detection. We found compelling uses for all of the transactional hash trie's innovations.

Use case – Fast non-transactional accesses

One of the most heavily used *LockFactory* methods is `zeroDurationlockObject`, which has the same semantics as acquiring and then immediately releasing a database lock. This method is called for each row visited by an SQL `join` or `select` in Derby's default isolation level of `TRANSACTION_READ_COMMITTED`. The read-committed isolation level requires only that all observed data has been committed, but it does not require that a query return results consistent with any single point in time.

```
378 class STMLockFactory {
379   ...
380   override def zeroDurationlockObject(
381       space: CompatibilitySpace, ref: Lockable,
382       qual: AnyRef, timeout: Int): Boolean = {
383
384     // try to get by without an atomic block
385     grantsByRef.single.get(ref) match {
386       case None => return true
387       case Some(grants0) => {
388         val id = LockIdentity(ref, space, qual)
389         if (compatibleWithGrants(grants0, id))
390           return true
391         if (timeout == 0)
392           return false
393
394         // enqueuing a waiter needs an atomic block, so try again
395         atomic { implicit txn =>
396           grantsByRef.get(ref) match {
397             case None => true
398             case Some(grants) if ((grants ne grants0) &&
399                                   compatibleWithGrants(grants, id)) => true
400             case _ => enqueueWaiter(id, null, true)
401           }
402         } match {
403           case z: Boolean => z
404           case w: Waiter => tryAwait(w, timeout) == null
405         }
406       }
407     }
408   }
409 }
```

Figure 4.16: Code to acquire and then immediately release a Derby lock.

Our STM implementation has a mutable transactional map that associates rows to the set of granted locks. This grant set is immutable, so we can test if a database lock may be immediately granted by fetching the grant set and then performing a local computation on the returned immutable reference. This means that in the common case `zeroDurationLockObject` can complete with only a single access to the *TMap*, so it can use the non-transactional fast path. If `zeroDurationLockObject` needs to wait then our implementation starts a transaction so that it can atomically check that it hasn't been canceled and enqueue itself onto the list of waiters. Figure 4.16 shows our implementation of this functionality.

Use case – STM integration

Our `STMLockFactory` relies heavily on the STM. We use transactions to atomically enqueue a waiter after detecting that a lock acquisition cannot be granted immediately, to atomically update both the global and the per-group multi-sets that record granted locks, to atomically process the waiter queue when releasing a lock, and to atomically check that a waiter hasn't been cancelled when granting it a lock. The atomic block in Figure 4.16 is an example of atomically checking if waiting is necessary and enqueuing a record describing the waiter.

Use case – Snapshots

Deadlock detection involves searching for cycles in which each of the members of the cycle is obstructed by its predecessor. This can be solved by a simple depth-first search that takes as input the set of active locks and the set of waiters. In Derby's fine-grained locking implementation, however, those sets are encoded in mutable data structures that are concurrently updated. It is not desirable to block the entire system when performing deadlock detection, so the existing implementation uses a two-phase locking strategy and a non-trivial proof that the DFS will converge. Despite these complications any database locks that are examined by the deadlock detector cannot be acquired or released until the cycle detector has completed.

`STMLockFactory` takes advantage of the transactional hash trie's efficient support for snapshots, and the ability to compose snapshot operations inside an atomic block. Prior to deadlock detection our implementation uses an atomic block to take a consistent snapshot of both the set of granted locks and the set of waiters. This makes the cycle detection code much simpler, because it is isolated from concurrent updates. It minimizes the impact on the rest of the system as well, because the cycle detector does not block other `LockManager` operations.

4.6.5 Experimental evaluation of STM inside Derby

Derby's fine-grained `LockManager` implementation represents our performance and scalability target. This code has been tuned and exercised in a wide variety of production

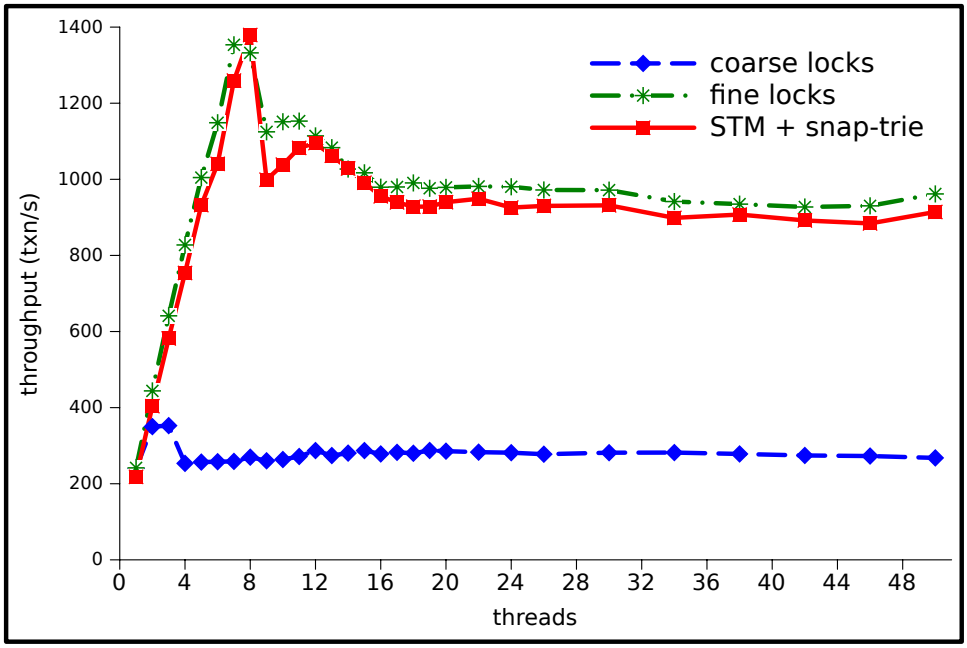


Figure 4.17: Derby performance for a 10,000 × 1,000 row join.

environments, and the additional engineering costs of concurrency have already been paid. Our STM-based implementation is substantially simpler; if we can match the fine-grained solution's execution behavior then we will consider ourselves successful.

Our in-situ experimental evaluation uses the test driver that the Derby developers used to isolate and evaluate the scalability limitations of Derby's original coarse-grained lock manager. This consists of a client and embedded database executing in a single JVM, with multiple threads executing SQL statements.

SQL joins

The most problematic case for the coarse-grained lock manager was heavy read access to the database. Figure 4.17 shows Derby's throughput with concurrent SQL joins. Each thread executes transactions that consist of a single inner join between a table with 10,000 rows and a table with 1,000 rows. The query result contains 1,000 rows.

The most striking result from Figure 4.17 is that the original lock manager implementation severely limits the end-to-end scalability of the database. Even though the underlying machine as 8 cores and 16 hardware thread contexts, parallelism provides at most a 47% throughput boost over a single-threaded test configuration. This scalability limit is especially disappointing because this workload consists entirely of reads, so there is no conceptual reason why concurrent queries should interfere with each other.

Derby's fine-grained `LockManager` implementation provides much better scalability. Its peak parallel throughput is 5.6 times its sequential throughput, occurring at 8 threads. It achieves this scalability without compromising single-threaded performance; its throughput when queried sequentially is essentially identical to that of the coarse-grained implementation. Figure 4.17 does not, however, reflect the additional complexity and engineering costs of this solution.

The STM implementation of `LockManager` also solves the scalability problem of the original implementation. Our implementation using the transactional hash trie provides a peak throughput that is 102% of the fine-grained version, and averages 95% of the performance of the fine-grained lock manager across all thread counts. These results are produced by code that is less than 1/5 as long, that performs deadlock detection without obstructing other threads, and that doesn't require sophisticated reasoning to tolerate inconsistent views of the system.

SQL updates

During the SQL join, most of the lock operations are zero-duration locks used to implement the read-committed SQL isolation level. In the common case these operations don't need to update any shared state or take advantage of the hash trie's composability. To test a scenario that stresses hash trie mutations and composition, we configured the Derby test client to measure the throughput of transactions that perform an SQL update. Each update acquires a lock from the `LockFactory` and releases it at the end of the database transaction. Figure 4.18 shows the throughput of the test client as a varying number of threads perform concurrent database updates.

The SQL update test stresses the databases I/O and durable logging subsystems. The test machine was configured with a pair of 1 terabyte 7,200 RPM drives configured with

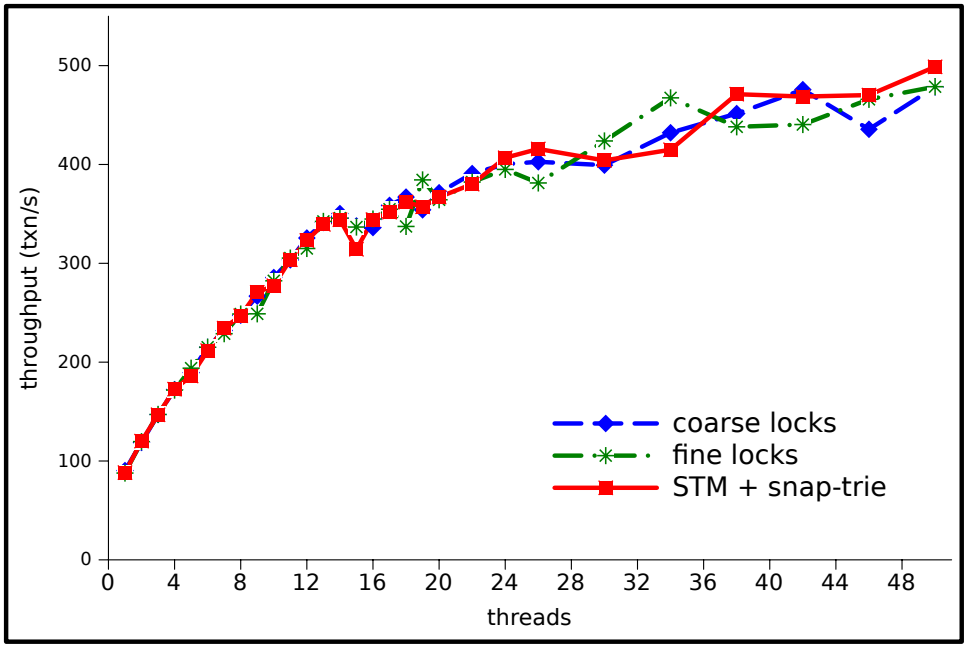


Figure 4.18: Derby performance for single-row updates in a 100,000 row table.

hardware RAID-0, controlled by an LSI Logic SAS1068E PCI-Express SCSI controller, and the test database was created on an ext3 filesystem.

Figure 4.18 shows that the coarse-grained lock manager is not the scalability limit for SQL updates. Fewer rows are locked and more work is performed per row. While reads can be served without I/O if they hit in the database's buffer cache, writes must always be passed through to the disks. For the update-heavy scenario this set of experiments show that our STM lock manager provides essentially the same throughput (0.5% more) than the fine-grained locking implementation.

4.7 Conclusion

The transactional hash trie introduced in this chapter combines the copy-on-write snapshots from Chapter 2 with the performant transactional integration of Chapter 3. It adds these powerful features while retaining good performance when they are not needed, as demonstrated by both microbenchmarks and an in-situ experimental evaluation. We achieve these

results with a novel strategy: adding transactional support to a linearizable lock-free algorithm.

Chapter 5

Conclusion

Data structures are defined by the primitive operations they support. Good design involves carefully selecting these primitives to balance several competing goals:

- The set of primitives must be complete for the problem domain;
- A smaller set of primitives is easier to understand; and
- The set of primitives must leave enough flexibility for an efficient implementation.

5.1 Loss of Composability

So long as execution contexts are isolated from each other, new high-level operations can be built by calling multiple primitives in sequence. This allows a small set of native operations to handle many novel application needs. For example, it is easy to implement an `incrementValue` method for a map by reading the value v associated with a key and then writing $v+1$. The available primitives of `get` and `put` are sufficient to efficiently implement the new functionality.

Unfortunately, the fundamental ability to compose primitives is lost when we move to shared memory multi-threading. Without external concurrency control, the native operations of a sequential mutable data structure can no longer be used to build new operations.

Existing concurrent sets and maps partially address this problem by adding the ability to compare-and-set the value associated with a key. This primitive allows the caller to build

an optimistic retry loop that performs an arbitrary transformation to the value associated with a key. Existing concurrent collections also add weakly consistent iteration. (Confusingly, the weakly consistent iteration primitive reuses the name of the strongly consistent iteration primitive that is available only in the non-thread-safe collection.) Despite limited composability, these data structures have seen wide use in multi-threaded programs.

STM provides automatic composability for the primitive operations of a data structure, restoring our ability to use a small set of primitives to cover a large problem domain. Unfortunately, simply adding STM to an existing algorithm results in a substantial performance penalty for all operations. This makes our job as designer much more difficult, because there is such a large tradeoff between efficiency and functionality.

Consistent iteration is especially problematic for an STM-based set or map. Long read-only transactions limit throughput except for multi-version STMs, but multi-version STMs have even higher constant overheads for all operations.

5.2 Our Solution

Our thesis is that we can provide concurrent sets and maps whose primitives are both efficient and composable, that we can have our cake and eat it too. We accomplish this with three contributions:

- A linearizable `clone` primitive can be added to concurrent tree-based data structures by using copy-on-write, and this primitive can be used to efficiently support consistent iteration;
- STM-specific algorithms minimize the performance penalty normally imposed by generic optimistic concurrency control; and
- STM can add composability to a lock-free algorithm, without requiring the overhead of transactions when no composition is desired.

Our proof by demonstration is ScalaSTM's *`TMap`*. This linearizable unordered map provides consistent iteration, composability for all of its operations, and excellent performance and scalability for a broad range of use cases.

Bibliography

[1] G. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 145, pages 263–266, 1962. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259âĂŞ-1263, 1962.

[2] M. Ansari, C. Kotselidis, M. Lujan, C. Kirkham, and I. Watson. On the performance of contention managers for complex transactional memory benchmarks. In *ISPDC '09: Proceedings of the 8th International Symposium on Parallel and Distributed Computing*, pages 83–90, 2009.

[3] P. Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.

[4] L. Ballard. Conflict avoidance: Data structures in transactional memory. Brown University Undergraduate Thesis, 2006.

[5] M. Bauer, J. Clark, E. Schkufza, and A. Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia. In *PPoPP '11: Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2011.

[6] R. Bayer. Symmetric binary B–Trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[7] R. Bayer and M. Schkolnick. Concurrency of operations on B–Trees. In *Readings in Database Systems (2nd ed.)*, pages 216–226, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[8] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[10] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 81–91, New York, NY, USA, 2007. ACM.

[11] L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. ResearchReport RR1998-18, LIP, ENS Lyon, March 1998.

[12] N. G. Bronson. CCSTM. http://github.com/nbronson/ccstm.

[13] N. G. Bronson. ScalaSTM. http://nbronson.github.com/scala-stm.

[14] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–268, New York, NY, USA, January 2010. ACM.

[15] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: High-performance concurrent sets and maps for STM. In A. W. Richa and R. Guerraoui, editors, *PODC*, pages 6–15. ACM, 2010.

[16] N. G. Bronson, H. Chafi, and K. Olukotun. CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days 2010*, April 2010.

[17] N. G. Bronson, C. Kozyrakis, and K. Olukotun. Feedback-directed barrier optimization in a strongly isolated stm. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–225, New York, NY, USA, 2009. ACM.

[18] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 56–67, New York, NY, USA, 2007. ACM.

[19] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[20] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *OOPSLA*, pages 835–847. ACM, 2010.

[21] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to domain parallelism. In *PPoPP '11: Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2011.

[22] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[23] L. Devroye. A note on the average depth of tries. *Computing*, 28:367–371, 1982. 10.1007/BF02279819.

[24] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208. Springer, March 2006.

[25] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, New York, NY, USA, 2009. ACM.

[26] A. DragojeviÄĞ, R. Guerraoui, and M. KapaÅĆka. Dividing transactional memories by zero. In *In Transact '08: 3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[27] R. Ennals and R. Ennals. Software transactional memory should not be obstruction-free. Technical report, 2006.

[28] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Memory - sequoia: programming the memory hierarchy. In *SC*, page 83. ACM Press, 2006.

[29] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC '09: Proceedings of the 23rd International Symposium on Distributed Computing*, pages 93–107. Springer, 2009.

[30] K. Fraser. *Practical Lock Freedom*. PhD thesis, University of Cambridge, 2003.

[31] E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, September 1960.

[32] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 258–264, New York, NY, USA, 2005. ACM.

[33] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[34] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[35] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Oct. 1978.

[36] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[37] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13, New York, NY, USA, October 2004. ACM Press.

[38] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architecture Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998.

[39] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[40] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, August 2007.

[41] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In D. Malkhi, editor, *DISC*, volume 2508 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2002.

[42] M. Herlihy and E. Koskinen. Transactional Boosting: A methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, USA, 2008. ACM.

[43] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *OPODIS '06: Proceedings of the 10th International Conference On Principles Of Distributed Systems*, December 2006.

[44] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[45] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

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

[47] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[48] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM New York, NY, USA, 2008.

[49] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC '06: Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 82–91, New York, NY, USA, 2006. ACM.

[50] *Java Specification Request (JSR) 133: Java Memory Model and Thread Specification*, September 2004.

[51] S. L. P. Jones. Harnessing the multicores: Nested data parallelism in haskell. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, page 138. Springer, 2008.

[52] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March–April 2005.

[53] E. Koskinen and M. Herlihy. Concurrent non-commutative boosted transactions. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[54] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[55] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

[56] B. C. Kuszmaul. The startech massively parallel chess program. *Journal of the International Computer Chess Association*, 18:117–140, 1995.

[57] K. S. Larsen. AVL trees with relaxed balance. In *IPPT '94: Proceedings of the 8th International Symposium on Parallel Processing*, pages 888–893, Washington, DC, USA, 1994. IEEE Computer Society.

[58] D. Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[59] D. Lea. Concurrent hash map in JSR 166 concurrency utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`, December 2007.

[60] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, 1998.

[61] E. Meijer, B. Beckman, and G. Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 706–706, New York, NY, USA, 2006. ACM.

[62] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPOPP '91: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, New York, NY, USA, 1991. ACM.

[63] V. Menon, S. Balensieger, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *SPAA '08: Proceedings of the 20th ACM Symposium on Parallel Algorithms and Architectures*, 2008.

[64] M. M. Michael. Hazard Pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel Distributed Systems*, 15(6):491–504, 2004.

[65] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *12th International Conference on High-Performance Computer Architecture*, February 2006.

[66] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 51–62, New York, NY, USA, 2008. ACM.

[67] J. E. B. Moss. Open nested transactions: Semantics and support. In *Poster at the 4th Workshop on Memory Performance Issues (WMPI-2006)*. February 2006.

[68] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–78, New York, NY, USA, 2007. ACM Press.

[69] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *PODS '87: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 170–176, New York, NY, USA, 1987. ACM.

[70] B. Pfaff. Performance analysis of BSTs in system software. *SIGMETRICS Performance Evaluation Review*, 32(1):410–411, 2004.

[71] A. Prokopec, T. Rompf, P. Bagwell, and M. Odersky. A Generic Parallel Collection Framework. Technical report, 2010.

[72] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, 1989.

[73] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.

[74] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[75] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 246–257, Washington, DC, USA, 2008. IEEE Computer Society.

[76] M. Roesch. Re: Pipelining and flowpinning. `http://seclists.org/snort/2010/q2/708`.

[77] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. Technical report, 2010.

[78] B. Saha, A. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO '06: Proceedings of the International Symposium on Microarchitecture*, 2006.

[79] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT–STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, March 2006. ACM Press.

[80] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, October 2008. ACM.

[81] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[82] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. S. III. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In P. Zhou, editor, *ICS*, pages 277–288. ACM, 2008.

[83] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing isolation and ordering in stm. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[84] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1), 2009.

[85] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel Distributed Computing*, 65(5):609–627, 2005.

[86] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[87] T. Sweeney. The next mainstream programming language: A game developer's perspective. Keynote at POPL'06, 2006.

[88] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *IEEE International Solid-State Circuits Conference (ISSCC 2008)*, San Francisco, CA, February 2008.

Appendices

Appendix A

Code for Transactional Predication

A.1 Reference Counting Code

Reference counting is used to reclaim unused predicates in a transactionally predicated map, as described in Section 3.4.2.

```
410 class Pred[V](@volatile var refCount: Int
411     ) extends TVar[Option[V]](None) {
412   def refCountCAS(v0: Int, v1: Int): Boolean = { ... }
413 }
414 class THashMap_RC[K,V] {
415   def get(k: K): Option[V] = {
416     val p = enter(k)
417     txn.afterCompletion({ _ => exit(k, p, 1) })
418     p.stmRead()
419   }
420   def put(k: K, v: V): Option[V] = {
421     val p = enter(k) // enqueue exit after delta known
422     val prev = try {
423       p.stmReadAndWrite(Some(v))
424     } catch {
425       // STM rollback is encoded as an exception
426       case x => exit(k, p, 1); throw x
427     }
428     if (prev.isEmpty) // no exit on successful insert
429       txn.afterRollback({ _ => exit(k, p, 1) })
430     else
```

```
431        txn.afterCompletion({ _ => exit(k, p, 1) })
432      prev
433    }
434    def removeKey(k: K): Option[V] = {
435      val p = enter(k)
436      val prev = try {
437        p.stmReadAndWrite(None)
438      } catch {
439        case x => exit(k, p, 1); throw x
440      }
441      if (!prev.isEmpty)
442        txn.afterCompletion({ t =>
443            exit(k, p, if (t.isCommitted) 2 else 1) })
444      else
445        txn.afterCompletion({ _ => exit(k, p, 1) })
446      prev
447    }
448
449    private val preds = new ConcurrentHashMap[K,Pred[V]]
450    private def enter(k: K) = enter(k, preds.get(k))
451    private def enter(k: K, p: Pred[V]): Pred[V] = {
452      if (p != null) {
453        val rc = p.refCount // p is stale if rc == 0
454        if (rc > 0) {
455          if (p.refCountCAS(rc, rc+1)) return p
456          return enter(k, p) // try again on p
457        }
458        preds.remove(k, p)
459      }
460      val fresh = new Pred[V](1)
461      val repl = preds.putIfAbsent(k, fresh)
462      if (repl == null) return fresh
463      return enter(k, repl) // try again on repl
464    }
465    private def exit(k: K, p: Pred[V], delta: Int) {
466      var rc = 0
467      do { rc = p.refCount
468      } while (!p.refCountCAS(rc, rc - delta))
469      if (rc - delta == 0) preds.remove(k, p)
470    }
471  }
```

A.2 Soft Reference Code

Code for managing predicates using soft references, as described in Section 3.4.3. *TokRef*.cleanup() is called from a daemon thread.

```
472 class Token {}
473 class TokRef[K,V](preds: ConcurrentMap[K,Pred[V]],
474     key: K, t: Token) extends CleanableRef[Token](t) {
475   var pred: Pred[V] = null
476   def cleanup(): Unit = preds.remove(key, pred)
477 }
478 class Pred[V](val softRef: TokRef[_,V]) extends
479     TVar[(Token,V)]((null, nullV)) {
480   softRef.pred = this
481 }
482 class THashMap_Soft[K,V] {
483   def get(key: K) = predForKey(key).stmRead()
484   def put(key: K, v: V): Option[V] = predForKey(key).stmReadAndWrite(Some(v))
485   def remove(key: K): Option[V] = predForKey(key).stmReadAndWrite(None)
486   private val preds = new ConcurrentHashMap[K,Pred[V]]
487   private def predForKey(k: K) = {
488     val p = preds.get(k)
489     if (p == null) createP(k) else enterPred(k, p)
490   }
491   private def enterPred(k: K, p: Pred[V]) = {
492     val t = p.softRef.get
493     if (t != null) {
494       Txn.current().addStrongRef(t)
495       (p,t)
496     } else {
497       preds.remove(k, p)
498       createPred(k)
499     }
500   }
501   private def createPred(k: K): (Pred[V],Token) = {
502     val t = new Token
503     val fresh = new Pred(new TokRef(preds, k, t))
504     val p = preds.putIfAbsent(k, fresh)
505     if (p == null) (fresh,t) else enterPred(k, p)
506   }
507 }
```

A.3 Unordered Iteration Code

Adding optimistic iteration to a transactionally predicated unordered map by counting insertions and removals, as described in Section 3.5.1. To minimize conflicts between writers, the counts are striped across multiple memory locations.

```
508 class TMap[K,V] {
509   private def stripes = 16
510   private val insertCts = Array.fromFunction(_ => { new TVar(0) })(stripes)
511   private val removeCts = Array.fromFunction(_ => { new TVar(0) })(stripes)
512   ...
513   def size: Int = {
514     insertCts.map(_.stmRead()).reduceLeft(_ + _) -
515         removeCts.map(_.stmRead()).reduceLeft(_ + _)
516   }
517
518   private def stripeIncr(a: Array[TVar[Int]]): Unit = {
519     val c = a(Thread.currentThread.hashCode() & (stripes - 1))
520     c.stmWrite(c.stmRead() + 1)
521   }
522
523   def put(key: K, value: V): Option[V] = {
524     val prev = predForKey(key).stmReadAndWrite(Some(value))
525     if (prev == None) stripeIncr(insertCts)
526     prev
527   }
528
529   def remove(key: K): Option[V] = {
530     val prev = predForKey(key).stmReadAndWrite(None)
531     if (prev != None) stripeIncr(removeCts)
532     prev
533   }
534
535   def elements = new Iterator[(K,V)] {
536     private val iter = {
537       for (c <- insertCts) c.stmRead()
538       predicates.entrySet().iterator
539     }
540     private var avail = step()
541
```

```
542     def hasNext: Boolean = avail != null
543     def next(): (K,V) = { val z = avail; avail = step(); z }
544
545     private def step(): (K,V) = {
546       if (!iter.hasNext()) {
547         null
548       } else {
549         val e = iter.next()
550         e.getValue.stmRead() match {
551           case Some(v) => (e.getKey, v)
552           case None => step()
553         }
554       }
555     }
556   }
557 }
```