[Parallel programming] is hard.

*Teen Talk Barbie*
(apocryphal)

# Chapter 1

# Introduction

How can we fully utilize the coming generation of parallel systems? The primitives available to today's programmers are largely inadequate. *Transactions* have been proposed as an alternative, but current implementations are either too limited or too inefficient for general use. This dissertation presents the design and implementation of efficient and powerful transaction systems to help address the challenges posed by current trends in computing hardware.

## 1.1 The rising challenge of multicore systems

Processor technology is finally bumping against its limits: even though transistor quantities continue to grow exponentially, we are now unable to effectively harness those vast quantities of transistors to create speedier single processors. The smaller transistors yield relatively slower signal propagation times, dooming attempts to create a single synchronized processor from all of those resources. Instead, hardware manufacturers are providing tightly integrated *multicore* systems, which integrate multiple parallel processors on one chip.

The widespread adoption of parallel systems creates problems: how can we ensure that operations occur in the appropriate order? Equivalently, how

can we ensure certain operations occur *atomically*, so that other components of the parallel system only observe data structures in well-defined states?

Atomicity in shared-memory multiprocessors is conventionally provided via mutual-exclusion ***locks*** (see, for example, [70, p. 35]). Although locks are easy to implement using test-and-set, compare-and-swap, or load-linked/store-conditional instructions, they introduce a host of difficulties. Protocols to avoid deadlock when locking multiple objects often involve acquiring the locks in a consistent linear order, which may make programming with locks error-prone and introduce significant overheads. The granularity of each lock must also be explicitly chosen: locks which are too fine introduce unnecessary space and time overhead, while locks which are too coarse sacrifice attainable parallelism (or may even deadlock). Every access to a shared object must hold some lock protecting that object, regardless of whether another thread is actually attempting to access the same object.

## 1.2 Advantages of transactions

*Transactions* are an alternative means of providing concurrency control. A transaction can be thought of as a sequence of loads and stores performed as part of a program which either ***commits*** or ***aborts***. If a transaction commits, then all of the loads and stores appear to have run atomically with respect to other transactions. That is, the transaction's operations are not interleaved with those of other transactions. If a transaction aborts, then none of its stores take effect and the transaction may be restarted, with some mechanism to ensure forward progress.

By structuring concurrency at a high level with transactions, human programmers no longer have to manage the details required to ensure atomicity. Programmers have difficulty correctly understanding the global interactions between multiple threads, each holding its own set of locks, and a full mental model of the global concurrency structure must be kept in mind when writing synchronization code. The simpler "global atomicity" guaranteed

under the transactional model eliminates errors and simplifies the conceptual model of the system, making future modifications safer as well.

The transaction primitives presented in this thesis can exploit optimistic concurrency, provide fault tolerance, and prevent delays by using non-blocking synchronization. Although transactions can be implemented using mutual exclusion (locks), our algorithms will utilize non-blocking synchronization [48, 36, 38, 52, 26] to exploit optimistic concurrency among transactions. Non-blocking synchronization offers a number of advantages; from the system builder's perspective the foremost is fault tolerance. A process that fails or pauses while holding a lock within a critical region can prevent all other processes from making progress. It is in general not possible to restore locked data structures to a consistent state after such a failure. Non-blocking synchronization offers a graceful solution to this trouble as non-progress or failure of any one thread will not affect the progress or consistency of other threads or the system.

Implementing transactions using non-blocking synchronization offers performance benefits as well. When mutual exclusion is used to enforce atomicity, page faults, cache misses, context switches, I/O, and other unpredictable events may result in delays to the entire system. Non-blocking synchronization allows undelayed processes or processors to continue to make progress. Similarly, in real-time systems, the use of non-blocking synchronization can prevent *priority inversion* in the system [43], although naïve implementations may result in starvation of low-priority tasks (see chapter 8.2 for a discussion).

## 1.3 Unlimited transactions

The ***Transactional memory*** abstraction [45, 35, 68, 60, 66, 34], has been proposed as a general and flexible way to allow programs to read and modify disparate primary memory locations atomically as a single operation, much as a database transaction can atomically modify many records on disk.

*Hardware transactional memory* (HTM) supports atomicity through (architectural) means, whereas *software transactional memory* (STM) supports atomicity through languages, compilers, and libraries. I will present both software and hardware implementations of the transaction model.

Researchers of both HTM and STM commonly express the opinion that transactions need never touch many memory locations, and hence it is reasonable to put a (small) bound on their size [35, 34].[1] For HTM implementations, they conclude that a small piece of additional hardware—typically in the form of a fixed-size content-addressable memory and supporting logic—should suffice. For STM implementations, some researchers argue additionally that transactions occur infrequently, and hence the software overhead would be dwarfed by the other processing done by an application. In contrast, this thesis will assume that transactions may be of arbitrary size and duration, and that details of the implementation should not be exposed to the programmer of the system.

My goal is to make concurrent and fault-tolerant programming easier, without incurring excessive overhead. This thesis discusses unbounded transactions because neither programmers nor compilers can easily cope when an architecture imposes a hard limit on transaction size. An implementation might be optimized for transactions below a certain size, but must still operate correctly for larger transactions. The size of transactional hardware should be an implementation parameter, like cache size or memory size, which can vary without affecting the portability of binaries.

In chapter 7 I will show how a fast hardware implementation for frequent short transactions can gracefully fail over to a software implementation designed to efficiently execute large long-lived transactions. The hybrid approach allows more sophisticated transaction models to be implemented,

[1] For example, [35, section 5.2] states, "Our [HTM] implementation relies on the assumption that transactions have short durations and small data sets"; while the STM described in [34] has quadratic slowdown when transactions touch many objects: performance is $O((R+W)R)$, where $R$ and $W$ are the number of objects opened for reading and writing, respectively.

while allowing a simpler hardware transaction mechanism to provide speed in the common case.

## 1.4 Strong atomicity

Blundell, Lewis, and Martin [12] distinguish between ***strongly atomic*** transaction systems, which protect transactions from interference from "non-transactional" code, and ***weakly atomic*** transaction systems which do not afford this protection. However, all current software transaction systems[2] are weakly atomic, despite the pitfalls thus opened for the unwary programmer, because of the perceived difficulty in efficiently implementing the required protection. Include more.

Strong atomicity is clearly preferable, and Blundell et al. point out that programs written for a weakly atomic model (to run on a current software transaction system, say) may deadlock when run under strong atomicity (for example, on a hardware transaction system). The transaction systems considered in this dissertation will preserve the correct atomic behavior of transactions even in the face of unsynchronized accesses from outside the transaction.

## 1.5 Summary of contributions

In summary, this thesis makes the following contributions:

- We provide efficient implementations of *strongly-atomic* transaction primitives to enable their general use. In particular, our technique imposes as little as 15% overhead on non-transactional code in a software-only system.
- The transaction primitives presented in this thesis can exploit optimistic concurrency, provide fault tolerance, and prevent delays by

[2] For example, [29].

using non-blocking synchronization.

- Unlike some previous work, this thesis will assume that transactions may be of arbitrary size and duration, and that details of the implementation should not be exposed to the programmer of the system.

- I will present both software and hardware implementations of the transaction model. The software transaction system runs real programs written in Java; I will discuss the practical implementation details encountered. The hardware transaction systems requires only small changes to the processor core and cache.

- I will show how a fast hardware implementation for frequent short transactions can gracefully fail over to a software implementation designed to efficiently execute large long-lived transactions.

In the next chapter I will provide some concurrent programming examples which illustrate the limitation of current lock-based methodologies. I will also provide examples illustrating the uses (some novel) of a transaction system, and conclude with a brief caution about the current limits of transactions.

In chapter 3, I will present the design of an efficient software-only implementation of transactions. Software-only transactions can be implemented on current hardware, and can easily accomodate many different transaction and nesting models. Software transactions excel at certain long-lived transactions, where the overhead is small compared to the transaction execution time. I will conclude by presenting a microbenchmark which demonstrates the performance limits of the design.

In chapter 4, I will discuss the practical implementation of chapter 3's design. I will present details of the compiler analyses and transformations performed, as well as solutions to problems which arise when implementing Java. I will then present benchmark results using real applications, discuss

the benefits and limitations revealed, and describe how the limits could be overcome.

In chapter 5 we consider one such limitation in depth, describing how large arrays fit into an object-oriented transaction scheme. We present a potential solution to the problem based on fast functional arrays.

In chapter 6, I will present hardware which enables very fast short transactions. The transaction model is more limited, but short committing transactions may execute with no overhead. The additional hardware is small and easily added to current processor and memory system designs.

At the end of the chapter, I present an *hybrid* transaction implementation, which builds on the strengths of simple hardware support, while allowing software fallback to support a robust and capable transaction mechanism. Unlike the extended hardware scheme, the transaction model is still easy to change and update; the hardware primarily supports fast small transactions and conflict checking in the common case. In the following chapter, we discuss some ways compilers can further optimize software and hybrid transaction systems. These opportunities may not be available to pure-hardware implementations.

In chapter 8, we discuss remaining challenges to the use of ubiquitous transactions for synchronization, and present some ideas toward solutions. Chapter 9 discusses related work, and our final chapter summarizes our findings and draws conclusions.

You have a hardware or a software problem.

*Service manual for Gestetner 3240*

# Chapter 2

# Examples of Transactional Programming

Before diving into the design of an efficient transaction system, I will motivate the transactional programming model by presenting four common scenarios which are needlessly difficult using lock-based concurrency; I will then present four novel applications which a transactional model facilitates. To ground the discussion in reality, I conclude by enumerating a few cases where transactions may *not* be the best solution.

## 2.1 Four old things you can't (easily) do with locks

Locks engender fragile and complex safety protocols which are often expressed externally to their implementations, poor modularity and composability, and an inability to deal gracefully with asynchronous events. These limitations of locks are well-known [37].

### 2.1.1 Tweak performance with localized changes

Preventing deadlocks and races requires global protocols and non-local reasoning. It is not enough to simply specify a lock of a certain granularity protecting certain data items; the order or circumstances in which the lock

may be acquired and released must also be specified globally, in the context of all locks in the system, in order to prevent deadlocks or unexpected races. This prevents the programmer from easily tuning the system using localized changes: every small change must be re-verified against the protocols specified in the whole-program context in order to prevent races and/or deadlock.

Furthermore, this whole-program protocol is not typically expressed directly in code: with common programming languages acquire/release ordering and guarantees must be expressed externally, often as comments in the source code which easily drift out of sync with the implementation. For example, in [4] we counted the comments in the Linux filesystem layer, and found that about 15% of these relate to locking protocols; often describing global invariants of the program which are difficult to verify. Many reported kernel bugs involve races and deadlocks.

### 2.1.2 Atomically move data between thread-safe containers

Another common programming pitfall with locks is their ***non-composability***. For example, given two thread-safe container classes implemented with locks, it is impossible to safely compose the *get* function of one with the *put* function of the another to create an atomic move. We must peek inside the implementation of the containers to synthesize an appropriate locking mechanism for such an action; for example, to acquire the appropriate locks on *both* containers (which may be at container, element, or some other level) before attempting the operation – and even then, we need to resort to some global lock ordering to guard against deadlock. Modularity must be broken in order to synthesize the appropriate composed function, which may be impossible.

In the Jade programming language, Martin Rinard presented a partial solution using "implicitly synchronized" objects [62, p14]. Lock acquisition for each module is exposed in the module's API as an executable "access declaration." Operation composition is accomplished by creating an access

declaration for the composed operation which invokes the appropriate access declarations for the components. The runtime system orders the lock acquisitions to prevent deadlock. This process suffices for conservative mutual exclusion, but it is limited in its ability to express alternative operation orders which preserve atomicity of the composed operation while overlapping its components.

Transactions do not suffer from the composability problem [30]. Because transactions only specify the atomicity properties, not the locks required, the programmer's job is eased and implementations are free to order operations in any way which preserves atomicity.

### 2.1.3 Create a thread-safe double-ended queue

Herlihy suggests creating a thread-safe double-ended queue using locks as "sadistic homework" for computer science students [37]. Although double-ended queues are a simple data structure, creating a scalable locking protocol is a non-trivial exercise if one wants dequeue and enqueue operations to complete concurrently when the ends of the queue are "far enough" apart, while safely handling the interference in the small-queue case. In fact, the solution to this assignment was a publishable result, as Michael and Scott demonstrated in 1996 [54].

The simple "one lock" solution to the double-ended queue problem, ruled out as unscalable in the locking case, is scalable and efficient for non-blocking transactions.

### 2.1.4 Handle asynchronous exceptions

Properly handling asynchronous events is difficult with locks, because it is impossible to safely go off to handle the event while holding an arbitrary set of locks—and it is impossible to safely drop the locks. The solution implemented in the Real-Time Specification for Java and similar systems is to generally forbid asynchronous events within locked regions, allowing the

programmer to explicitly specify certain points within the region at which execution can be interrupted, dropping all locks in order to do so. Maintaining the correctness in the face of even explicitly-declared interruption points is still difficult.

Transactional atomic regions handle asynchronous exceptions gracefully: the transaction is aborted to allow an event to occur.

## 2.2 Four new things transactions make easy

I present three examples in this section, illustrating how transactions can support fault tolerance and backtracking, simplify locking, and provide a more intuitive means for specifying thread-safety properties. I will first examine a destructive traversal algorithm, showing how a transaction implementation can be treated as an exception-handling mechanism. I then, using a network flow example, show how this transaction mechanism can be used to simplify the locking discipline required when synchronizing concurrent modifications to multiple objects. Finally, I show an existing race in the Java standard libraries (in the class `java.lang.StringBuffer`). "Transactification" of the existing class corrects this race.

### 2.2.1 Destructive traversal

Many recursive data structures can be traversed without the use of a stack using pointer reversal. This technique is widely used in garbage collectors, and was first demonstrated in this context by Schorr and Waite [65]. An implementation of a pointer-reversal traversal of a simple singly-linked list is shown in Figure 2.1.

The `traverse()` function traverses the list, visiting nodes in order and then reversing the `next` pointer. When the end of the list is reached, the reversed links are traversed to restore the list's original state.

Of course, I have chosen the simplest possible data structure here, but the technique works for trees and graphs—and the reader may mentally