Atomicity for Concurrent Programs
Outsourcing Report

By Khilan Gudka <kg103@doc.ic.ac.uk>
Supervisor: Susan Eisenbach

June 23, 2007

Contents

Chapter 1

Introduction

Commodity computing is rapidly shifting from single-core to multi-core architectures, as CPU manufacturers find that traditional ways of increasing processor performance are no longer sustainable. While parallel processing has been around for a long time, it is only recently that it is becoming so widespread. Intel predicts that by the end of 2007, 90% of the processors that it ships will be multi-core [81].

In order to best utilise such parallel processing, software needs to be concurrent. This is a significant leap given that the vast majority of programs are sequential [87], that is they have a single thread of control and can only perform one logical activity at any one time. Concurrent programs on the other hand are structured using several threads of execution, enabling multiple activities to be performed simultaneously. For example, a web server can accept and process multiple client requests at the same time.

Previously with single-core systems, concurrent programs did not provide many performance advantages as true parallelism was not possible. Hence, the main reason for writing software in this way was for convenience, better throughput and increased responsiveness. However, now the tables are turned. Concurrent programming is the way that programmers will get increases in performance from their software, given that significant increases in clock speeds are no longer likely [87, 89].

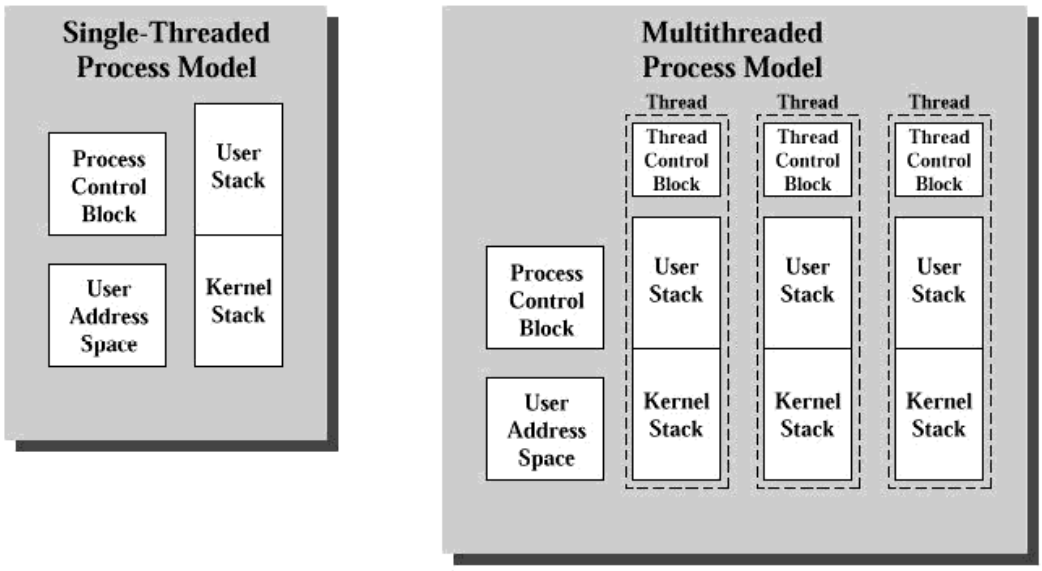
Automatic parallelisation of sequential programs offers one possible alternative that would enable existing software to take advantage without modification [8, 15]. However, many researchers agree that these automatic techniques have been pushed to their limits and can exploit only modest parallelism. Instead, programs themselves should be restructured to become logically more concurrent [62].

Hence, the industry-wide shift towards parallel architectures has and will continue over the next few years to spark a revolution in concurrent programming, making it a pervasive aspect of software engineering [88].

1.1 The subtleties of concurrent programming

Concurrent programming goes all the way back to the late 1950s, when time sharing systems were introduced to make better use of expensive computing resources. This involved multiplexing a single CPU between different users enabling them to work on the same machine at the same time doing different things. Up until the 1990s, it remained mainly of concern to operating systems programmers whom had to ensure that their operating systems made optimal use of available hardware

Figure 1.1: *Concurrent programs consist of multiple threads of execution, each of which has its own stack and CPU state (stored in the Thread Control Block). They reside within an operating system process and communicate with each other through shared memory. (Image source: http://www.cs.cf.ac.uk/Dave/C/thread_stack.gif).*



resources and allowed the user to multi-task. However, over the last decade or so it has been made more and more accessible to common programmers with modern programming languages such as Java [35] providing explicit language support for threads, and older languages adding support through libraries [75].

However, even though there is such widespread support for concurrency, programmers are still very reluctant to write concurrent code. While concurrency has not provided much performance benefit, the main reason why programmers tend to avoid it is because *concurrent programming is inherently difficult and error-prone* [76].

Concurrent programs consist of multiple threads of execution that reside within an operating system process. Each thread has its own stack and CPU state, enabling them to be independently scheduled. Moreover, in order to facilitate communication (threads are designed to work together), they share some of the process's address space (Figure 1.1). However, this "common" memory is the root cause of all problems associated with concurrent programming. In particular, if care is not taken to ensure that such shared access to memory is controlled, it can lead to interference (more commonly referred to as a *race condition* [74]). This occurs when two or more threads access the same memory location and at least one of the accesses is a write.

Figure 1.2 shows an example race condition whereby two threads T1 and T2 proceed to increment a `Counter` object `c` concurrently by invoking its `increment` method. This method reads the value of the counter into a register, adds 1 to it and then writes the updated value back to memory. Figure 1.2(c) shows an example interleaving. Thread T1 reads the current value of the counter (0) into a register but is then pre-empted by the scheduler which then runs thread T2. T2 reads the value (0) into a register, increments it and writes the new value (1) back to memory. T1 still thinks that the counter is 0 and hence when it is eventually run again, it will also write the value 1, overwriting the update made by T2. This error is caused because both threads are allowed uncontrolled access to shared memory, i.e. there is no synchronisation. As a result, a race condition occurs and an

Figure 1.2: *An example race condition that occurs when two threads T1 and T2 proceed to increment a counter at the same time.*

```
class Counter {
  int counter = 0;

  void increment() {
    counter = counter + 1;
  }
}

Counter c = new Counter();
Thread T1: c.increment();
Thread T2: c.increment();
```

(a)

increment() execution steps:

read *counter* into register;
add 1 to register;
write register to *counter*;

(b)

	Thread T1	Thread T2
1	*counter* is 0	
2	read *counter* into register	
3		*counter* is 0
4		read *counter* into register
5		add 1 to register
6		write register to *counter*
7		*counter* is 1
8	add 1 to register	
9	write register to *counter*	
10	*counter* is 1	

(c)

update is lost.

Such interference can be extremely difficult to detect and debug because their occurrence depends on the way the operations of different threads are interleaved, which is non-deterministic and can potentially have an infinite number of possible variations. As a result, they can remain unexposed during testing, only to appear after the product has been rolled out into production where it can potentially lead to disastrous consequences [64, 58, 77].

1.2 Preventing race conditions

At present, programmers prevent such race conditions by ensuring that conflicting accesses to shared data are mutually exclusive, typically enforced using locks. Each thread must acquire the lock associated with a resource before accessing it. If the lock is currently being held by another thread, it is not allowed to continue until that thread releases it. In this way, threads are prevented from

Figure 1.3: *Race free version of the example given in Figure 1.2.*

```
class Counter {
  int counter = 0;

  synchronized void increment() {
    counter = counter + 1;
  }
}

Counter c = new Counter();
Thread T1: c.increment();
Thread T2: c.increment();
```

performing conflicting operations at the same time and thus interfering with each other.

Figure 1.3 shows a race-free version of our counter example. The `synchronized` keyword is Java syntax that requires the invoking thread to acquire first an exclusive lock on the object before proceeding. If the lock is currently unavailable, the requesting thread is blocked and placed in a queue. When the lock is released, it is passed to a waiting thread which is then allowed to proceed. Going back to our example, now thread T2 will not be allowed to execute `increment` until T1 has finished because only then can it acquire the lock on `c`. Thus, invocations of `increment` are now serialised and races are prevented.

1.2.1 The complexities of using locks

The mapping from locks to objects and whether or not non-conflicting operations can proceed in parallel is up to the programmer. This is referred to as the locking granularity and can have significant repercussions on performance. Some examples include:

1. Single global mutual exclusion lock (that is, a global lock that can be held by only one thread at a time) to protect all shared accesses of all objects.
2. Mutual exclusion lock for each object (e.g. `synchronized` in Java) that subsequently prevent multiple threads from accessing the same object at the same time but which permit concurrent accesses to different objects.
3. Separate read and write locks for each object that allow non-conflicting accesses on the same object to proceed in parallel. This makes it possible for several threads to read the value of the counter concurrently but only one thread is allowed access when updating it.

One of the problems with this technique of concurrency control is that it's imperative—the programmer has to enforce it. It can therefore be an easy source of errors, especially when locking policies[1] are complex (such as (3) above). If under-synchronising occurs, then the possibility of race conditions still remains, violating safety. Programmers also have to ensure that locks are acquired in the correct order otherwise deadlock may occur, leading to a progress violation.

[1]The locking policy is the strategy used for acquiring/releasing locks.

Figure 1.4: *An example of deadlock. (a) is an extended version of the `Counter` class from Figure 1.3 with an `equals` method to check if the current `Counter` has the same value as a second `Counter` object. Moreover, threads T1 and T2 execute this method on two separate instances, passing the other instance as the argument. (b) shows an example resulting locking schedule that leads to deadlock. Note that like race conditions, the occurrence of deadlock also depends on the order in which operations are interleaved.*

```
class Counter {
  int counter = 0;

  synchronized void increment() { ... }

  synchronized boolean equals(Counter c) {
    synchronized(c) {
      return counter == c.counter;
    }
  }
}

Counter c1 = new Counter();
Counter c2 = new Counter();

Thread T1: c1.equals(c2);
Thread T2: c2.equals(c1);
```

(a)

	T1	T2
1	lock c1	
2		lock c2
3		lock c1
4	lock c2	waiting
5	waiting	waiting

(b)

Figure 1.4 extends our counter example with an `equals` method that compares two `Counter` objects for the same value. Before this method accesses the second counter, it must first acquire a lock on it to ensure interference does not occur. Thus, it must acquire both a lock on the counter whose `equals` method has been invoked and the counter we are comparing with it. However, if another thread tries to acquire these locks in the opposite order (as shown in Figure 1.4(b)), then deadlock may result.

Note how the actual occurrence of deadlock in the example depends on the way operations are interleaved. This is similar to race conditions, however deadlocks are easier to debug because the affected threads come to a standstill. Another problem illustrated by the example is that modularity must be broken in order to detect where deadlock may occur. Therefore, methods can no longer be treated as black boxes and must be checked to ensure that locks are not acquired in a conflicting order (although a number of tools exist that can statically check for deadlocks by building a lock graph and then look for cycles [6]).

The possibility of deadlock can be eliminated by making the locking granularity coarser, so that a single lock is used for all `Counter` objects. However, this has a negative effect on performance as non-conflicting operations, such as incrementing different counters, would not be allowed to proceed in parallel. Hence, hitting the right balance can be difficult. Furthermore, consider if the `Counter` class were part of a library. A static analyser might detect that there is a possibility of deadlock, but how can you prevent it? You would need to ensure that `c1.equals(c2)` and `c2.equals(c1)` are not called concurrently by synchronising on another lock. However, this just adds to the complexity!

Other problems that can occur due to locks include:

- **Priority inversion:** occurs when a high priority thread T_{high} is made to wait on a lower priority thread T_{low}. This is of particular concern in real-time systems or systems that use spin-locks (that busy-wait instead of blocking the thread) because in these, T_{high} will be run in favour of T_{low}, and thus the lock will never be released. Solutions include raising the priority of T_{low} to that of T_{high} (priority inheritance protocol) or the highest priority thread in the program (priority ceiling protocol) [59].
- **Convoying:** can occur in scenarios where multiple threads with similar behaviour are executing concurrently (e.g. worker threads in a web server). Each thread will be at a different stage in its work cycle. They will also be operating on shared data and thus will acquire and release locks as and when appropriate. Suppose one of the threads, T, currently possesses lock L and is pre-empted. While it is off the CPU, the other threads will continue to execute and effectively catch up with T up to the point where they need to acquire lock L to progress. Given that T is currently holding this lock, they will block. When T releases L, only one of these waiting threads will be allowed to continue (assuming L is a mutual exclusion lock), thus the effect of a convoy will be created as each waiting thread will be resumed one at a time and only after the previous waiting thread has released L [23, 93].
- **Livelock:** similar to deadlock in that no progress occurs, but where threads are not blocked. This may occur when spin-locks are used.

Thus, concurrent programming with locks introduces a lot of additional complexity in the software development process that can be difficult to manage. This is primarily because current techniques are too low-level and leave the onus on the programmer to enforce safety and liveness properties. As a result, even experts can end up making mistakes [57].

Lock-free programming [31] is one alternative that allows multiple threads to update shared data concurrently in a race-free manner without using locks. Typically this is achieved using special atomic update instructions provided by the CPU, such as Compare-and-Swap (CAS) and Load Linked/Store Conditional (LL/SC). These update a location in memory atomically provided it has a particular value (in CAS this is specified as an argument to the instruction, while for LL/SC it is the value that was read using LL). A flag is set if the update was successful, enabling the program to loop until it is. The new `java.util.concurrent` framework [61] introduced in Java 5 SE provides high-level access to such atomic instructions, making lock-free programming more accessible to programmers.

While lock-free algorithms avoid the complexities associated with locks such as deadlock, priority inversion and convoying, writing such algorithms in the first place can be even more complicated. In fact, lock-free implementations of even simple data structures like stacks and queues are worthy of being published [47, 30]. Thus, such a methodology doesn't seem like a practical solution in the short run.

1.3 The quest for better abstractions

Given the problems associated with current abstractions and that programmers face an inevitable turn towards concurrency, a lot of work is currently being done to find ways of making concurrent programming a lot more transparent. Some advocate that we need completely new programming languages that are better geared for concurrency, but given that we don't yet know exactly what this means, they suggest this shift should be gradual [87].

Many have proposed race-free variants of popular languages that perform type checking or type inference to detect if a program contains races [11, 19, 38], while others abstract concurrency into the compiler enabling programmers to specify declaratively their concurrency requirements through compiler directives [90]. Alternative models of concurrent computation have been suggested such as *actors* [2] and *chords* [9] as well as a number of flow languages that enable programmers to specify their software as a pipeline of operations with parallelism being managed by the runtime [12, 56].

However, these proposals either require programmers to change substantially the way they write code or they impose significant overheads during development such as the need to provide annotations. This limits their practicality and usefulness in the short-term.

1.4 Race-freedom as a non-interference property

Ensuring that concurrent software does not exhibit erroneous behaviour due to thread interactions has traditionally been interpreted as meaning that programs must be race-free. However, race-freedom is not sufficient to ensure the absence of such errors.

Figure 1.5 shows a bank account example to illustrate this. The program consists of two threads T1 and T2 that concurrently update `Account` objects `a1` and `a2`, both of whose balance is initially £10. T1 is transferring £10 from `a2` to `a1` and T2 is making a withdrawal from `a2` for the same amount. T1 executes first and reads the balance of account `a2`. It asserts that `a2` has sufficient funds for the transfer but is then pre-empted. T2 then proceeds to withdraw £10 from `a2` and is run to completion by the scheduler. Sufficient funds still exist and therefore the withdrawal completes successfully. The balance for account `a2` is now £0. Meanwhile, T1 is resumed but remains unaware of this change and thus still thinks that the transfer can go ahead. It proceeds to withdraw £10 from `a2` (which has no effect) and then deposits £10 for 'free' into account `a1`, creating an inconsistency.

Such incorrect behaviour occurred due to thread T2 being able to modify account `a2` while the transfer was taking place. This was possible because although the invocations of `getBalance` and `withdraw` ensure mutually exclusive access to account `a2`, their composition does not. As a result, conflicting operations can be interleaved between them and thus lead to higher-level interferences. This does not introduce races because all methods in the `Account` class are `synchronized` and therefore acquire the correct locks before performing shared accesses.

1.5 Enter the world of atomicity

In order to assert that such interferences do not occur, we need a stronger non-interference property that ensures that other threads cannot interleave conflicting operations while a block of code is executing, that is the *atomicity of code blocks*. A code block S is said to be atomic if the result of any concurrent execution involving S is equivalent to an execution without any interleavings. Thus it appears to other threads to execute in "one step" which incidentally is what programmers mostly intend when using concurrency control mechanisms such as locks [29]. However, race detection tools do not allow programmers to assert that such a high-level property actually holds.

Atomicity is a very powerful concept, as it enables us to reason safely about a program's behaviour at a higher level of granularity. That is, it abstracts away the interleavings of different threads (even though in reality, interleaving will still occur), enabling us to think in terms of single-threaded semantics. Hence, it provides a *maximal* guarantee of non-interference.

Figure 1.5: *Bank account example illustrating that race-freedom is not a strong enough non-interference property. That is, errors caused by thread interactions can still occur even if a program has no races. (a) is the race free `Account` class and a description of two threads T1 and T2. (b) shows an example interleaving that leads to an error.*

```
class Account {
  int balance = 0;

  public Account(int initial) {
    balance = initial;
  }

  synchronized int getBalance() {
    return balance;
  }

  synchronized int withdraw(int amount) {
    if(amount >= balance)
      amount = balance;

    balance -= amount;
    return amount;
  }

  synchronized void deposit(int amount) {
    balance += amount;
  }

  synchronized void transferFrom(Account from, int amount) {
    int fromBalance = from.getBalance();
    if(amount <= fromBalance) {
      from.withdraw(amount);
      deposit(amount);
    }
  }
}
```

```
Account a1 = new Account(10);
Account a2 = new Account(10);

Thread T1: a1.transferFrom(a2,10);
Thread T2: a2.withdraw(10);
```

(a)

	T1	T2
1	a2.getBalance()	
2		a2.withdraw(10)
3	a2.withdraw(10)	
4	a1.deposit(10)	

(b)

1.5.1 Verifying atomicity

Over the last few years, a number of techniques have been developed for verifying atomicity properties of arbitrary blocks of code, including the use of type systems [29, 28, 27], dynamic analysis [26, 91], theorem proving [33] and model checking [46].

1.5.1.1 Type Systems

Approaches based on type systems are by far the most popular and can be split up into type checking and type inference. Type checking typically involves requiring the programmer to annotate which blocks of code are intended to be atomic and then using Lipton's reduction theory [65] to show that any concurrent execution of such blocks can be serialised (transformed into an equivalent execution with no interleavings), taking into account the acquisition and release of locks. This is achieved using the concept of a *mover*:

- **Left mover:** An action `a` is a left mover if whenever `a` follows *any* action `b` of another thread, the actions `a` and `b` can be swapped without changing the resulting state. For example, lock release operations.
- **Right mover:** An action `a` is a right mover if whenever `a` is followed by *any* action `b` of another thread, they can be swapped without changing the resulting state. For example, lock acquire operations.
- **Both mover:** An action that is both a left mover and a right mover. For example, accessing an object while holding a mutual exclusion lock on it.

The relevance of this is that if it can be shown that a code block S consists of 0 or more right movers followed by at most one atomic action followed by 0 or more left movers, then it is possible to transform a concurrent execution involving S into a serial one by repeatedly commuting operations. Furthermore, given the generality of the definitions above, this would ensure that S is atomic in any possible interleaving. A program 'type checks' if all blocks marked atomic can be reduced in this way. Figure 1.6 shows an example reduction of the `increment` method from the `Counter` class in Figure 1.3 (with synchronisation having been made more explicit and an intermediate step introduced). E1, E2 and E3 are interleaved expressions of other threads.

Type inference techniques differ in that they instead infer the atomicity of a code block based on annotations provided by the programmer indicating which locks guard each data item. The atomicity inferred can be one of a number of different levels such as constant (i.e. pure), atomic and compound (i.e. not atomic). Type inference techniques can also complement type checking where the programmer supplies an expected atomicity.

1.5.1.2 Theorem Proving and Model Checking

These approaches are similar to those based on type systems in that they require the programmer to provide atomicity specifications for methods/blocks of code as well as annotations indicating which locks protect which objects. However, they differ in how they verify atomicity, namely using theorem proving and model checking respectively. Furthermore, they are not as popular, due to their inability to scale to large programs. In particular, they suffer from the problem of state-space explosion and thus are only suitable for use in cases where the number of states is small, such as for unit testing.

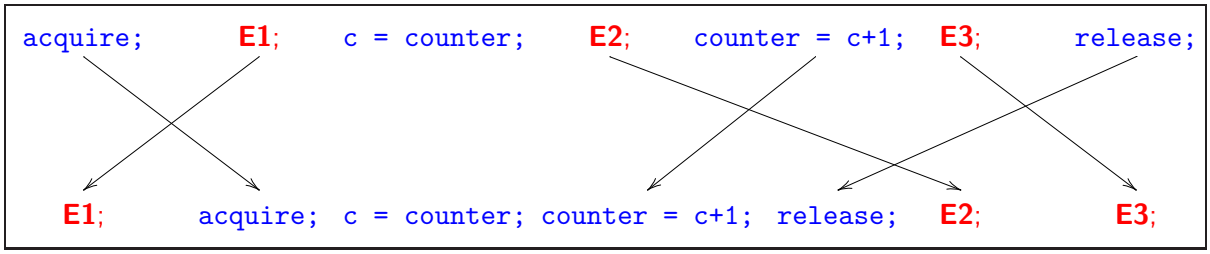
Figure 1.6: *Verifying atomicity using reduction. (a) is the* `Counter` *class from Figure 1.3 with* `increment` *having been slightly modified so synchronisation is more explicit and that there is an intermediate step to model better what actions would really happen when this method is executed. Furthermore, each operation is labelled with the type of mover it is. (b) shows the actual process of reducing an arbitrary interleaving into a serial execution by repeatedly commuting, thus showing that* `increment` *is atomic.*

```
class Counter {
  int counter = 0;

  void increment() {
    synchronized(this) { // lock acquire (right mover)
      int c = counter;   // both mover
      counter = c + 1;   // both mover
    }                    // lock release (left mover)
  }
}
```

(a)

Verifying that increment() is atomic using reduction:
(E1, E2 and E3 are interleaved operations of other threads)



(b)

1.5.1.3 Dynamic Analysis

Dynamic analysis verifies atomicity properties at runtime instead of at compile-time. While this has the disadvantage that only a limited number of execution paths are checked (also known as *test case coverage*), it avoids much of the annotation overhead imposed by static analyses and can scale better when analysing large programs. However, dynamic analysis does incur a runtime overhead, and given the nature of atomicity as a safety property and the potential ramifications of atomicity violations, one might argue that verification should really be performed at compile-time.

1.6 Atomicity: an abstraction?

Atomicity provides a maximal non-interference guarantee to the programmer, ensuring that if a block of code can be verified to have this property, then it can be assumed that any concurrent execution involving it will be equivalent to a serial execution. However, while this makes reasoning

Figure 1.7: *An atomic implementation of method* `transferFrom` *from the* `Account` *class of Figure 1.5.*

```
synchronized void transferFrom(Account from, int amount) {
  synchronized(from) {
    int fromBalance = from.getBalance();
    if(amount <= fromBalance) {
      from.withdraw(amount);
      deposit(amount);
    }
  }
}
```

about program behaviour much simpler, the onus is still on the programmer to enforce it. That is, *writing concurrent software is still inherently complicated*. For example, consider the steps required to make `transferFrom` atomic: Being a `synchronized` method, no other thread can access the current account until execution completes and the lock is released. However, to ensure that no other thread accesses account `from` during the transfer, its lock must be acquired and held throughout. This prevents all but the transferring thread from accessing the two accounts involved and thus the execution of method `transferFrom` is now atomic. Figure 1.7 gives the updated version.

However, the improved implementation (and ensuring atomicity in general) has introduced another problem, namely the potential for deadlock. This could occur if transfers between two accounts are performed in both directions at the same time. In fact, this seems to be the way that atomicity would be enforced in general, given that one would have to ensure that other threads cannot access the shared objects throughout. Thus, the problems mentioned previously with regards to locks still exist and it is still the programmer that has to deal with them.

Furthermore, it may not always be possible to ensure atomicity. Consider if instead we were invoking a method on an object that was an instance of some API class. Acquiring a lock on this object may not be sufficient if the method accesses other objects via instance fields, as we would need to acquire locks on those objects too in case they are accessible from other threads. However, accessing those fields would break encapsulation and might not even be possible if they are private. Hence, the only solution would be for the class to provide a `Lock()` method that locks all its fields. However, this *breaks abstraction* and *reduces cohesion* because now the class has to provide operations that are not directly related to its purpose.

Hence, although we can now more confidently assert the absence of errors due to thread interactions by verifying that a block of code is atomic, programmers are still responsible for ensuring it. With current abstractions, this may not even be possible due to language features such as encapsulation. In fact, even if it is possible, modularity is broken thus increasing the complexity of code maintenance, while other problems such as deadlock are also increasingly likely. Thus, we are desperately in need of better abstractions that alleviate the programmer from needing to deal with such complexities and which simplify writing concurrent code.

1.7 Atomic sections

1.7.1 The garbage collection analogy

Until 10 years ago, most programming languages supported manual memory management whereby the programmer explicitly takes care of the allocation and deallocation of memory. However, entrusting the user with such an important task naturally led to the possibility of several major classes of bugs appearing in a program. For example, programmers have to ensure that memory is freed when no longer needed, otherwise it cannot be reused leading to the problem of memory leaks. Additionally, pointers need to be kept track of otherwise memory may be freed when it is still needed, resulting in dangling pointers which can lead to errors that are hard to diagnose. Moreover, trying to free memory through such pointers can result in heap corruption or even further dangling pointers if that memory has been reallocated.

To circumvent such problems, programmers need to be aware of which memory locations are being accessed where and carefully devise an allocate/deallocate protocol. This leads to breaking modularity as callers and callees must know what data the other may access. In short, it is extremely difficult [94, 39].

In order to relieve the programmer of such complexities, automatic garbage management [68] was introduced. This abstracts the deallocation (the programmer still has to allocate explicitly, although abstractly) of memory into the language implementation, eliminating a class of errors that had plagued software programmers for several decades.

1.7.2 A similar approach for concurrency

Verifying atomicity involves the programmer annotating those methods/blocks of code that are intended to be atomic and then using static or dynamic analysers to show that this is in fact the case. While this provides a maximal non-interference guarantee, it is still the programmer that has to ensure it. Moreover, ensuring atomicity at the source code level is not always possible.

This has led to a new kind of abstraction that pushes concurrency control and subsequently the enforcing of atomicity properties into the language implementation. Atomic sections [66] are blocks of code that execute as if in a single step, with the details of how this is achieved being taken care of by the programming language. This is a significant improvement over current abstractions as atomic sections completely relieve the programmer from worrying about concurrency control and thus eliminate the associated complexities. They enable programmers to think in terms of single-threaded semantics and thus also remove the need to make classes/libraries thread safe. Furthermore, error handling is considerably simplified because code within an atomic section is guaranteed to execute without interference from other threads and thus recovering from errors is like in the sequential case. They are also composable, that is two or more calls to atomic methods can be made atomic by wrapping them inside an atomic section. There is no need to worry about which objects will be accessed as all these details are taken care of by the language implementation. Therefore, they also promote modularity.

However, what makes them even more appealing is that they don't require the programmer to change the way he/she codes. In fact they simplify code making it much more intuitive and easier to maintain. Furthermore, recall that programmers intent is mostly atomicity when using locks [29], thus atomic sections enable programmers to more accurately specify their intentions. Figure 1.8 shows an implementation of the `Counter` class using atomic sections (denoted using the `atomic` keyword). Note that there is no longer the potential for deadlock as the invocation of `equals` is

atomic.

1.7.3 Popularity

The idea of atomic sections for programming languages is not new [66] but like automatic garbage collection, has only recently begun to generate intense interest from the programming community. In fact, it is one of the hottest topics in computing at the moment, with the likes of Microsoft, Intel, Sun and numerous academic institutions getting involved. Furthermore, a number of programming languages have already started to include support for them either as part of the language [16, 3, 13, 14] or through libraries [21, 31, 51, 48] and a growing list of language extensions have also been proposed in the literature [43, 44, 82].

1.7.4 Implementing their semantics

Atomic sections are quite an abstract notion, giving language implementors a lot of freedom in how they are implemented. A number of techniques have been proposed over the years including:

- **Interrupts:** Proposed in Lomet's seminal paper [66], whereby interrupts are disabled while a thread executes inside an atomic section. This ensures atomicity by preventing thread switches but doesn't work when you have multiple processors or if interrupt-driven I/O is being performed (in the atomic section).
- **Co-operative scheduling:** Involves intelligently scheduling threads such that their interleavings ensure atomicity [85].
- **Transactional memory:** This is the most popular technique which implements atomic sections as database style transactions. Memory updates are buffered until the end of the atomic section and are committed in 'one step' if conflicting updates have not been performed by other threads while it was executing, otherwise the changes are rolled back (i.e. the buffer is discarded) and the atomic section is re-executed [4, 21, 44, 43, 49, 55, 54, 60, 71, 79, 86].
- **Lock inferencing:** An interesting approach that automatically infers which locks need to be acquired to ensure atomicity and inserts the necessary synchronisation in such a way that deadlock is also avoided [18, 25, 53, 69].
- **Object proxying:** A very limited technique whereby proxy objects are used to perform lock acquisitions before object invocations at runtime. While they can ensure that the resulting concurrent interleavings are serialisable, they impose significant performance overheads, have very coarse granularity, and require the programmer to specify explicitly which objects are shared. In addition, object field accesses cannot be protected, while nested sections are also not supported properly. Implementations of atomic sections that use this technique are provided as a library and thus programmers have to `begin()` and `end()` the atomic section of code explicitly, even if an exception is thrown [24], Although a source-to-source translator could automate this process.
- **Hybrids:** Approaches that combine several of the above techniques. For example, one hybrid approach uses locks when there is no contention or when an atomic section contains an irreversible operation, and transactions otherwise [92].

While nobody yet knows what is the best way of implementing atomic sections, transactional memory seems to be the most popular approach. However, in its purest form, it is not expressive

Figure 1.8: *An implementation of the `Counter` class using atomic sections. Atomic sections group multiple statements together and execute them as if in one atomic step with the details of how this is achieved being taken care of by the language implementation. They relieve the programmer from having to worry about thread interactions and concurrency control, enabling him/her to think in terms of sequential semantics. Furthermore, the composition of atomic methods is made atomic simply by wrapping them inside an atomic section. Previously, the outcome of the code executed by threads T1 and T2 would depend on the order of thread interleavings, but with atomic sections one can reason about their behaviour sequentially.*

```
class Counter {
  int counter = 0;

  atomic void increment() {
    counter = counter + 1;
  }

  atomic boolean equals(Counter c) {
    return counter == c.counter;
  }
}

Counter c1 = new Counter();
Counter c2 = new Counter();

Thread T1:
  atomic {
    c1.increment();
    c2.increment();
    c1.equals(c2);  // guaranteed to return true + no deadlock
  }

Thread T2:
  atomic {
    c1.increment();
    c2.increment();
    c2.equals(c1);  // guaranteed to return true + no deadlock
  }
```

enough to accomodate the full range of operations that could occur in an atomic section (due to it being a memory abstraction) and has significant performance overheads in both contended and uncontended cases.

Lock inferencing is a promising alternative, as it overcomes many of the problems associated with transactions. However, there are still many subtle issues that need to be overcome for it to be applicable for a real language. There are partial solutions, but model languages are often too simple [53], or require annotations from the programmer [69]. Also, in some approaches, the number of locks is related to the size of the program [53] and thus they do not scale well when programs have a large number of objects. Furthermore, language features such as arrays, exceptions, subclassing and polymorphism have yet to be considered while issues such as aliasing also need a lot of work.

This project looks at building upon existing work in lock inferencing by addressing these issues, details of which will be looked at in the next section. To facilitate experimentation, a multi-threaded Java-like toy language called `singlestep` (see Appendix A for its grammar) will be implemented. While modifying a real language is a possible alternative, lock inferencing is still at a stage where a lot of conceptual challenges need to be solved.

This concludes the introduction, which has hopefully provided a pleasant taster of the inherent problems with concurrent programming given current abstractions and how the notion of atomicity and atomic sections will revolutionise the way we write concurrent code.

1.8 Report structure

The remainder of this report is structured as follows:

- **Background:** This next chapter takes a look at some of the implementation techniques mentioned in the previous section, with an emphasis on lock inferencing.
- **Specification:** A more detailed outline of the goals of the project.
- **Evaluation:** This chapter looks at how it can be determined that the aims of the project have been met.

Chapter 2

Background

This part of the report looks in more detail at some of the proposed techniques for implementing atomic sections, as mentioned in the introductory chapter. The aim is to explore their good and bad points and consider how they might affect the decisions made in this project. The project aims to build on work done in lock inferencing techniques, therefore more emphasis will be made in this area.

2.1 Terminology

In this section we look at terminology that is commonly found in the literature and which will be used throughout the rest of this report.

2.1.1 Strong vs. weak atomicity

Conceptually, atomic sections execute as if in 'one atomic step,' abstracting away the notion of interleavings. However, enforcing such a guarantee is not always entirely possible, due to limitations in hardware, the nature of the implementation technique or the performance degradation that would result. To make the particular atomicity guarantee offered by an implementation explicit, two terms have been defined in the literature [10]:

- **Strong atomicity:** the intuitive meaning of atomic sections as appearing to execute atomically to all other operations in the program regardless of whether they are in atomic sections or not.
- **Weak atomicity:** atomicity is restricted to be only with respect to other atomic sections.

For example, it may appear that transactional memory provides strong atomicity by default, but this is not necessarily the case due to the limitations of atomic read-modify-write instructions such as Compare-and-Swap (CAS), which software transactional memory relies upon. Recall that CAS instructions atomically update a memory location provided that it has some particular current value. However, they suffer from the 'ABA' problem whereby a memory update may be masked by a subsequent one that restores the previous value. Hence, it becomes possible for updates made by code outside transactions to go un-noticed when a transaction validates for thread interference and thus atomicity may be violated. Load-Linked/Store-Conditional (LL/SC) atomic update instructions can detect this 'hidden update,' however, they only support updating one location at a time. In particular, it is not possible first to load several values using LL, and then atomically update these locations later using SC, which would be required for transactional memory.

Figure 2.1: *A non-blocking implementation of condition variables that require races to ensure progress. The purpose of this example is to illustrate that wrapping non-atomic code inside* `atomic{ }` *to ensure strong atomicity shouldn't lead to synchronisation between operations that were designed to race. In this particular case, the invocations on the condition variable* `c` *made by threads T1 and T2 would need to be wrapped inside an atomic section to ensure that the correct semantics are enforced for thread T3. However, this has the side effect that T1 and T2 now cannot race on* `c.counter`

```
class Condition {

  boolean condition = false;

  void wait() {
    while(!condition) { }
    condition = false;
  }

  void notify() {
    condition = true;
  }
}

Condition c = new Condition();
```

```
Thread T1:
  for(int i=0; i<99; i++) { }
  c.notify();

Thread T2:
  c.wait();

Thread T3:
  atomic {
    c.condition = false;
  }
```

However, there are some implementations that get around this and are able to offer the intended behaviour [54]. Therefore, this introduces the problem that we now don't have a clear semantics for atomic sections. This has a negative effect on code portability because it can be shown that software written assuming one atomicity may break when presented with the other [10].

Ideally, atomic sections should provide strong atomicity as this is what programmers expect and is what makes them such a useful abstraction. However, the performance degradation that results from enforcing this may be too high thus representing a trade off between performance and ease of programming. Although a number of optimisations have been proposed for reducing this overhead [54].

It should be noted that providing strong atomicity doesn't mean that an implementation has to directly support it. In fact, an implementation may only provide weak atomicity but provide the guarantee of strong atomicity by using a static analysis to detect shared accesses outside atomic sections and subsequently wrap them inside `atomic{}`. However, care has to be taken here as it may have the side-effect of preventing races that the program relies upon for progress (as is illustrated in Figure 2.1).

2.1.2 Closed nested vs. open nested transactions

For composability, it is important that atomic sections support nesting. This can trivially be achieved by considering nested atomic code to be part of the outermost section, however unnecessary contention can occur as a result. Furthermore, it may be necessary to communicate shared state out of an atomic section, such as for communication between threads. Consequently, a number of different nested semantics have been developed (note that they have been designed in the context of

transactional memory) [72]:

- **Closed nested:** In this approach, each nested transaction executes in its own context, that is, it performs its own validation for the locations it has accessed. If a nested transaction commits, that is, no other thread has performed conflicting updates to the locations it has accessed, then the changes are *merged* with the parent's read/write set. This has the advantage that conflicts are detected earlier and only requires rolling back the transaction at the current nesting level, although the outermost transaction will still need to validate these accesses in case another thread has performed a conflicting update before it reached the end. Other threads do not see the changes until the outermost transaction commits.

- **Open nested:** Closed nested transactions can still lead to unnecessary contention, given that updates made by child transactions are not propagated to memory until the end of the outermost transaction. As a result, another type of nesting semantic has been proposed, which actually makes the updates of a nested transaction visible to other threads. This has the advantage that it permits shared state to leave atomic sections, such as for communication between atomic sections, although it has the disadvantage that other threads may see inconsistent state if an outer transaction later aborts, requiring mechanisms such as locking [72] to overcome this. Furthermore, programmers must supply undo operations to undo the effects of the open nested transaction, given that simply restoring a log will not suffice as other threads may have performed updates in the mean time.

With regards to lock inferencing, we can only release locks when we are sure they are no longer required as we cannot roll back. Hence, we will most probably treat all nested atomic sections as part of the outermost one, although this can be significantly improved by various optimisations that will be described later (see Section 2.3).

2.2 Transactional memory

Transactional memory provides the abstraction of database-style transactions [22] to software programs, whereby a transaction in this context is a sequence of memory operations whose execution is serialisable or equivalently, has the properties of atomicity, consistency and isolation.[1] That is, each transaction either executes completely or it doesn't (atomicity), it transforms memory from one consistent state into another (consistency), and the result of executing it in a multi-threaded environment is equivalent to if the transaction was executed without any interleavings from other threads (isolation).

These semantics can be achieved in a number of different ways [21], although the predominant approach is to execute transactions using *optimistic concurrency control*. This is a form of non-blocking synchronisation in which transactions are executed assuming that interference will most probably not occur; that is, another thread is highly unlikely to write to locations that it accesses. To ensure atomicity, tentative updates are buffered during execution and committed atomically at the end. For isolation, this commit is only allowed to proceed if another transaction has not already performed a conflicting update. This typically requires storing the initial value for each location accessed and validating that they remain unchanged. If a conflict is detected, the tentative updates are discarded and the transaction is re-executed. Note that consistency automatically

[1] Transactions in database theory have the additional property of durability, although this is irrelevant here as we are concerned with interactions between threads that occur through main memory, which is volatile.

follows provided that the programmer has ensured that invariants would be maintained even if the transaction was executed in isolation.

Transactional memory provides a number of *potential* advantages over traditional blocking primitives such as locks, including:

- **No deadlock, priority inversion or convoying:** as there are no locks! Although, in theory a slightly different form of priority inversion could still occur if a high priority thread was rolled back due to an update made by a low priority thread.
- **More concurrency:** recall that with locks, the amount of concurrency possible is dependent on the locking granularity. However, as the number of locks increase, so does the complexity involved in managing them and thus programmers may end up settling for policies that afford sub-optimal levels of concurrency. Transactional memories provide the finest possible granularity (at the word level) by default, resulting in optimal parallelism. However, this comes at the cost of increased overheads, which are unnecessary when the number of concurrent atomic sections is low.
- **Automatic error handling:** Memory updates are automatically undone upon rollback, reducing the need for error handling code [41]. However, this is orthogonal to the topic of atomicity as atomic sections ensure sequential semantics, which is most important.
- **No starvation:** transactions are not held up waiting for blocked/non-terminating transactions, as they are allowed to proceed in parallel even if they perform conflicting operations.

However, these advantages rely on being able to roll back in the event of a conflict. This proves to be a huge limitation for atomic sections as it prevents them from containing *irreversible operations* such as system calls and most types of I/O. In addition, allowing conflicting transactions to proceed in parallel poses a problem for *large transactions* that may be repeatedly rolled back (livelock) due to conflicts with many smaller ones. Even in the general case, it leads to wasted computation when transactions are rolled back, not to mention the overheads incurred during logging and validation. A number of workarounds have been proposed, such as buffering I/O [42] and contention management [84], but no general solution exists yet.

In comparison, lock inferencing does not suffer from these problems because of its pessimistic nature. Nevertheless, transactional memory still seems to be the most popular technique for implementing atomic sections, with many hardware, software and hybrid implementations having been proposed. We now look at these in a bit more detail.

2.2.1 Hardware transactional memory (HTM)

The original proposal for transactional memory was a hardware implementation by Herlihy and Moss [49], whom showed that transactions could be supported using simple additions to the cache mechanisms of existing processors, and by exploiting existing cache coherence protocols. Their HTM executed transactions optimistically, keeping separate read and write sets for each transaction in a small transactional cache. However, it had the limitations that (1) it could only support transactions upto a fixed size (where size refers to the number of memory locations accessed) and (2) transactions could not survive scheduler pre-emption.

These limitations were due to there being a bounded amount of available transactional resources. As a result, many early HTMs were *best-effort* [60]. A best-effort HTM provides efficient support

for as many transactions as available resources allow, but does not guarantee to be able to commit transactions of any size or duration. However, these size and duration restrictions are highly architecture dependent, thus removing many of the software engineering benefits of transactions, as programmers have to make assumptions about hardware.

Hence, most recent work in HTMs has concentrated on providing support for larger or even unbounded transactions (both in terms of size and duration). Example techniques include, overflowing transactional state into a table allocated in memory by the operating system [4] and also into a thread's virtual address space [4, 79, 71]. However, as these data structures have to be traversed in hardware, the result is a more complicated HTM.

2.2.1.1 Conclusion

HTMs provide the advantage of superior performance in comparison to software implementations. However, their main limitation is that *they require architectural change*. Transactions in databases have been around for a long time and are in widespread use, yet we haven't seen hardware support being introduced to improve their performance. Thus proposals face the tough task of convincing chip manufacturers that HTMs are necessary and also relatively simple to add to their existing designs. This is complicated by the fact that they must support large/unbounded transactions, with current hardware-only designs being inherently complex.

The other problem is *portability*. Early proposals imposed architectural-dependent limitations; however, new hybrid approaches [60] improve things by providing an abstraction layer decoupling the underlying HTM from the program utilising hardware support when available otherwise transparently resorting to software transactional memory if not or if the HTM does not have sufficient resources. Such proposals also simplify the hardware design as HTMs only have to be best-effort.

HTMs are irrelevant for lock inferencing given that it doesn't use transactions, although a hybrid implementation could benefit from better performance with hardware support.

2.2.2 Software transactional memory (STM)

To overcome the limitation of requiring specialised hardware, Shavit and Touitou [86] proposed a software-variant called software transactional memory (STM). Recall that transactional memory was originally motivated by the need for easier and more efficient ways of implementing non-blocking synchronisation operations, as it was thought that the key to highly concurrent programming was to decrease the number and size of critical sections or even eliminate them by implementing programs as non-blocking [49, 86]. Consequently, Shavit and Touitou's initial STM and many other early implementations [31, 32, 52, 70] focused on being non-blocking.

However, recently it has been shown that such a guarantee is not necessary and by dropping it, significantly better performance can be achieved [21]. Hence, many newer STMs have omitted the non-blocking requirement and instead use a combination of optimistic synchronisation and locks [20, 21, 45] or only locks [55, 83] (although, it should be noted that the latter class of STMs still retain the need for transactions to be abortable, in order to dynamically avoid deadlock and starvation). This gives promising evidence that using locks for implementing atomic sections is definitely a step in the right direction.

STM is a very active area of research with a lot of progress having been made over the last few years. Other developments include object-based STMs [52, 5, 45], better support for nested transactions [73], customisable contention management [40, 52, 84], conflict-driven notification [44, 13] and improved support for I/O and exceptions [42, 41].

However even though there have been many advancements, the main focus has been on improving performance [45]. Hence, a lot more work still needs to be done to address issues hindering their practicality as an implementation mechanism for atomic sections. In this section, we look in a bit more detail at how STM research has evolved since 1995 and its implications as an implementation technique for atomic sections.

2.2.2.1 Word-based vs. Object-based STMs

Just as locks can protect data at the level of words or objects, STM implementations also differ in the granularity at which they detect contention. In *word-based STMs* [86, 43, 44], the unit of concurrency is an individual memory word. That is, contention is considered to occur when threads access the same location in memory. *Object-based STMs* [70, 32, 31, 52, 5, 55, 45] on the other hand are higher-level and see memory as being organised as a number of blocks (group of memory words) or objects. In these systems, contention is considered to occur when threads access the same block/object, even though they may be accessing different words within it.

Word-based STMs have the advantage that they are finer-grained and thus permit more parallelism than object-based ones. For example, they allow threads to update different fields of the same object concurrently. However, this typically incurs high overheads both in space and time, and also doesn't correspond very well with modern programming paradigms. Object-based STMs on the other hand are coarser, but as a result have less overheads and are easier to implement for languages with objects.

A significant advantage of object-based STMs is that they do not incur additional costs during reads and writes. This is because they typically clone objects before first accessing them and proceed with using the clone; thus, they can use normal read and write operations. Word-based STMs on the other hand, typically require searching a log on every read/write to obtain the most up-to-date value, which incurs huge overheads. However, to efficiently facilitate the cloning approach, a level of indirection is required for referencing objects so that it is possible to change which object a reference points to atomically (e.g. using CAS) when the transaction commits. Furthermore, while the cost of cloning small objects is not so bad, large objects pose a problem. Potential solutions include representing such objects as functional arrays [5].

Given that object-based STMs have less overheads, this is the most common type of STM found in the literature at present. Moreover, the above technique of cloning is the most typical approach used in object-based STMs [31, 52, 70], although other techniques such as maintaining lists of reading and writing transactions in each object have also been proposed [5].

2.2.2.2 Non-blocking STMs

As mentioned at the beginning of this section, initial STM implementations were non-blocking. In a non-blocking implementation, the suspension or failure of any number of threads cannot prevent the remainder of the system from making progress, thus providing robustness against poor scheduling decisions as well as arbitrary thread termination/failure [32]. Consequently, it prohibits the use of ordinary locks because, unless the thread that currently holds the lock continues to run, the lock can never be released and therefore the non-blocking semantics cannot be guaranteed. Instead, it relies upon the provision of special instructions, such as Compare and Swap (CAS) or Load Linked/Store Conditional (LL/SC) that can perform atomic updates on memory. For example, Figure 2.2 is a non-blocking implementation of the `Counter` class in Figure 1.3 that uses CAS. This instruction takes three arguments: the memory location to be updated, its expected value and the value to update it to. If the current value of `counter` is as expected, then it performs the update (atomically)

Figure 2.2: *A non-blocking implementation of the* `Counter` *class of Figure 1.3.*

```
class Counter {
  int counter = 0;

  void increment() {
    while(!CAS(&counter, counter, counter+1)) { }
  }
}
```

and returns true, otherwise it does nothing and returns false. In this way, it *tries* to ensure that the update is atomic.[2]

Non-blocking algorithms can be classified according to the kind of progress guarantee they provide [32]:

- **Obstruction-freedom:** This is the weakest form of progress assurance: a thread is only guaranteed to make progress so long as it does not contend with other threads for access to any location at the same time. This implies that threads which aren't running cannot prevent it from progressing, thus requiring that a transaction be able to roll back and retry. When there is contention however, it does not prevent the possibility of livelock, whereby a thread cannot progress because other threads keep getting into its way. The chance of this occurring is reduced using a contention manager, which determines what to do when contention for memory is detected. Policies include exponential back off and aborting the conflicting transaction [52]. In the case of back off, the contention manager ensures that a transaction is not backing off indefinitely by aborting the conflicting transaction after a threshold is reached. Note that this doesn't guarantee the absence of livelock as a transaction may repeatedly conflict with different transactions.

  Research shows that the choice of contention management policy is application-specific and can have a significant impact on performance [84].

- **Lock-freedom:** Adds the requirement that the system as a whole makes progress, even if there is contention. In some cases, lock-free algorithms can be developed from obstruction-free ones by adding a helping mechanism: if thread T2 encounters thread T1 obstructing it, then T2 helps T1 to complete T1's operation. For example, it may assist in committing T1's updates for it or yield the processor. Once that is done, T2 can proceed with its own operation and hopefully not be obstructed again. This is sufficient to prevent livelock, although it does not offer any guarantee of per-thread fairness [32, 31].

- **Wait-freedom:** Adds the requirement that every thread makes progress, even if it experiences contention. This gives a hard bound on the number of instructions that need to be executed to perform any operation and thus is the strongest non-blocking progress guarantee. However, it is seldom possible to develop wait-free algorithms that offer competitive practical performance [32].

[2] It cannot guarantee that the update is atomic, as updates by other threads that set the value to the expected value will go undetected.

Shavit and Touitou's initial STM was word-based and lock-free, using helping to achieve this. In their implementation, each transaction acquires ownership of all locations being accessed in it (specified up front by the programmer) before executing the body of the transaction. If a location has already been acquired by another transaction, it helps the conflicting transaction before releasing the locations it has already acquired and restarting. Each thread has an associated record which is used to store information about its current transaction, such as the memory locations being accessed, its current status and a number of other fields used to synchronise with threads that may help it.

Lock-free algorithms typically use recursive helping [31], however this can be costly in terms of performance [86]. This STM avoids recursive helping by ensuring that memory locations are acquired in order and by restarting transactions after they have helped a conflicting transaction. Consequently, it is much more efficient than traditional lock-free approaches [86], although it also has a number of disadvantages, including:

- **Static transactions:** Helping requires that locations are acquired in some global order, hence the programmer has to specify up front which memory locations are accessed in the transaction. This was deemed acceptable in the paper because STM was designed to make it easier to implement higher-level non-blocking synchronisation operations such as multi-word CAS (MCAS) [32], which require knowing the memory locations in advance anyway. However, this is not feasible in the general case, such as for traversing dynamic data structures where it is not known in advance which memory locations will be accessed. Furthermore, having to specify all memory accesses upfront also breaks modularity.

- **Memory overheads:** A vector, the same size as memory is required to hold information about which transaction owns the corresponding memory word. This indirection is typical of non-blocking approaches and is one of their disadvantages. Consequently, performance also suffers because additional cache misses will be incurred when reading a memory word. On the other hand, such fine granularity allows more parallelism.

- **Helping overhead:** The only justifiable need for helping is in case the thread executing the conflicting transaction has failed. This could be due to a hardware failure or a computer failing in the case of a distributed system. However, distributed applications are a niche and processor failures are extremely unlikely. Lock-free programs *have* to provide such mechanisms due to the guarantee they promise, but such assurances are not in general necessary for atomic sections [21].

On the other hand, Shavit and Touitou's STM has the advantage that it doesn't incur the overheads of logging present in many other STMs, given that threads are only aborted before acquiring ownership of all required memory locations. Nevertheless, the requirement for specifying accesses up front, the unnecessary overheads caused by helping and the memory cost make it undesirable.

Later non-blocking implementations include Moir's lock-free and wait-free STMs [70]. The lock-free version splits memory up into a fixed number of blocks, which form the unit of concurrency (object-based STM). It overcomes some of the limitations of the former STM such as the need to specify upfront which memory locations are accessed. However, it introduces additional drawbacks as a result. In particular, this approach uses *optimistic synchronisation* as described earlier and thus introduces the need for logging, with writes being performed on copies of blocks and version numbers being used to detect conflicts. This results in significant performance overheads due to searching the log on each access, validation, copying blocks, committing, etc. Reads can be especially expensive

Figure 2.3: *Example of opening an object before accessing it in object-based STMs [52]. Shared objects have to be encapsulated within wrapper objects to allow them to be changed atomically (a). To access the original object in a transaction, the wrapper must be 'opened' (b). This opening process may perform bookkeeping, acquisition and/or consistency checks. The specific things differ between STMs. For example, in DSTM, opening an object in write mode causes it to be acquired while in FSTM, a copy of it is added to the transaction's read-write list. Note that it is required that objects only keep references to these wrapper objects and not the original ones, otherwise it would be possible to bypass the transactional mechanisms.*

```
Counter counter = new Counter();
TMObject tmObject = new TMObject(counter);
```

(a)

```
Counter counter = (Counter)tmObject.open(WRITE);
counter.increment();
```

(b)

because incremental validation is performed (that is, the STM validates that the block being read from is still consistent on each read). The rationale for this is that if the block being read from has been updated by another thread, then the transaction is sure to fail and so should not carry on otherwise it could lead to a situation that would not otherwise occur in a serial execution of the transaction, such as memory access violations, infinite looping and arithmetic faults [67]. Other significant disadvantages include wasted computation performed by a transaction that is destined to abort. In STMs that only perform validation just before committing [52, 44], this is a big drawback, although in Moir's implementation validation is incremental and thus conflicts are detected earlier. Benchmarks show that how often validation should be performed is application-specific [67].

More recent non-blocking STMs include Fraser's FSTM [31, 32] and Herlihy et al's DSTM [52, 51]. These are both object-based and support dynamic transactions, however FSTM is lock-free and uses recursive helping, while DSTM is obstruction-free and uses contention management. Both clone an object before writing to them and thus require indirection for object references. This is achieved using wrapper objects that hold references to the real ones. In FSTM, this wrapper object is called an *object header* and simply holds a reference to the actual object, while in DSTM, it is called a `TMObject` and instead contains a reference to a `Locator` object, which in turn holds a reference to the descriptor of the transaction that last updated this particular object as well as the current and last versions of the object. The reason for this extra level of indirection will become clear later.

Before objects are accessed inside transactions, they have to be 'opened' (see Figure 2.3 for an example). An object can be opened in *read mode* or *write mode*. In both approaches, opening an object in read mode causes it to be added (just a reference to, not copy of) to the transaction's *read list*, while opening an object in write mode has differing semantics:

In DSTM, this results in acquiring the object. In particular, it creates a `Locator` object storing (1) a reference to this transaction, (2) the current value of the object and (3) a copy of it. It then uses CAS to automatically switch the current `Locator` object to this new one. If the transaction

that is being referenced by the current `Locator` object is still active, this means there is contention and subsequently a contention manager is queried for what to do (wait, abort, etc). FSTM on the other hand allows multiple transactions to optimistically write to the same object at the same time. Thus, it instead adds a copy of the object to a read-write list for the current transaction. Contention is not checked for until commit time because it must acquire objects in some global order to ensure that help cycles do not occur and thus must wait until all objects have been opened (upon trying to acquire an object already acquired by another transaction, the current transaction recursively helps the conflicting one before restarting). This is due to it being lock-free and consequently leads to significantly more wasted computation. On the other hand, DSTM requires an extra level of indirection for acquiring objects upon opening them and thus may experience slower reads and writes as a result. Although, acquiring objects instead of optimistically updating them means that at commit time, all the transaction needs to do is make sure that it hasn't been aborted.

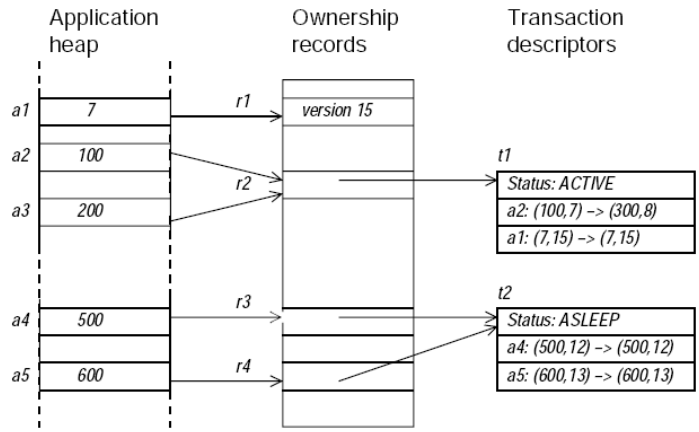
Nevertheless, both approaches still have to validate that what they have read is still consistent. This cannot be delayed till the end of the transaction, because objects may be modified by other threads while the current transaction is executing (as copies are not made for reads). This is of significance because it can lead to problems such as infinite looping, memory access violations and arithmetic faults [67]. Consequently, validation has to be performed on each open for reading, which is extremely expensive and is thus a significant problem with optimistic approaches [67]. Furthermore, with FSTM, ensuring that objects are acquired in order requires sorting their addresses before a commit. One alternative is to keep the read-write list sorted, although the overheads would then be incurred when inserting [67].

In summary, FSTM provides nice progress guarantees but requires that objects be acquired in order to prevent help cycles and thus has to support optimistic updates. Consequently, conflicts are not detected until the transaction commits, potentially leading to significantly more wasted computation and other overheads such as sorting. Furthermore, helping is only really necessary if a thread has failed, given that it can perform the updates itself if it hasn't. DSTM provides the weaker guarantee of obstruction-freedom and thus has a simpler and more efficient implementation. In particular, it can acquire objects before writing to them, thus removing the need for validating such objects, although it requires double indirection to achieve this. This has the downside of potentially slower reads and writes. Moreover, both have the disadvantage of requiring objects to be opened before accessing them plus the need for incremental validation, which has a significant impact on performance given that it is done whether there is contention or not. On the other hand, they don't require read/write barriers as found in word-based STMS [44, 43].

Harris and Fraser proposed an obstruction-free word-based STM [43] and were the first to consider using STMs for implementing atomic sections in modern object-oriented languages such as Java. Unlike Shavit and Touitou's STM that has an array of ownership records (orecs) the same size as memory, this STM uses a hash table of orecs whose size does not have to match that of memory (note that if the hash table is smaller, multiple locations will hash to the same orec). Figure 2.4 illustrates this organisation. An orec may hold a *version number* or a pointer to the *current owning transaction* for the locations that are associated with it (i.e. that hash to it). Version numbers are used to detect conflicts and must be incremented each time one of the associated memory words is updated.

The other kind of structure are *transaction descriptors* which store the current status of each active transaction and the memory accesses that it has made so far. Both reads and writes in this STM are optimistic, thus transaction descriptors keep track of addresses accessed, their old and new values and the old and new version numbers of those values (old values and versions are those before the transaction first accessed that particular orec, while the new values and versions are as a result

Figure 2.4: *Data structures in Harris and Fraser's word-based STM [43].*



of executing the current transaction so far). This imposes substantial overheads while reading and writing because firstly, the descriptor has to be searched each time for the latest values and secondly, version numbers have to be kept consistent. Note that multiple locations may share version numbers, thus when updating a version number in the transaction descriptor upon performing a write, the transaction also has to update all entries for locations that map to the same orec.

When the transaction completes executing, it attempts to commit by temporarily acquiring all orecs associated with the locations it has accessed. Acquisition involves installing a reference to the transaction's descriptor in these orecs and then changing the descriptor's status to `COMMITTED`, before actually writing the values to memory. However, this can only occur provided that the orec has the same version number as that in the transaction descriptor. If the version numbers differ or if the orec has already been acquired by another transaction, the commit fails, acquired orecs are released and the transaction is aborted. Further details can be found in [43].

This STM has a number of significant performance issues including the overheads of searching logs during each read/write, the overhead of determining version numbers/keeping version numbers consistent as well as the possibility of transactions that access disjoint memory locations contending with each other if they share orecs (see [43] for suggested improvements).

One interesting feature though of this paper is that the programmer can specify a entry condition that must be true before the atomic section is executed. That is, the general form of their atomic construct is: `atomic (condition) { statements }`. However, care has to be taken to ensure that a nested atomic section does not have a contradicting condition such as `n != 0` if the parent's condition is `n == 0` and `n` has not yet been modified by it.

2.2.2.3 Omitting the non-blocking requirement

Semantically, non-blocking programs are desirable because they provide specific progress guarantees, which make reasoning about them easier. However, this comes at the cost of implementation complexity and performance. Furthermore, such promises are often too strong, covering too wide a range of scenarios, whereas weaker guarantees would suffice in the general case. In fact, we are already seeing this trend as newer non-blocking STMs are forsaking the assurances of lock/wait-

freedom and instead settling for the weaker property of obstruction-freedom because it leads to simpler and more efficient implementations [43, 52, 50].

However, recent work suggests that even this weakest guarantee is a hindrance [21]. The main arguments for non-blocking STMs in the literature, aside from STMs originally being designed for use in non-blocking programs, include [21]:

- **Prevents long-running transactions from blocking others:** Non-blocking STMs allow conflicting threads to proceed in parallel and hence long transactions do not starve smaller ones. However, this argument is flawed because in order for a large transaction to be able to commit, no conflicts must occur while it is running. This would mean that conflicting transactions should be blocked otherwise the long transaction would never make progress.
- **Prevents the system locking up if a thread is switched out:** Some argue that the system may lock up when using locks if a thread is switched out while holding a lock. This isn't necessarily true because in the majority of cases, the thread will eventually be switched in again. We say the majority, because it is possible for a thread to be blocked waiting for I/O which never comes, although the probability of this happening is low.
- **Fault tolerance:** When using locks, if a thread fails, it may not release ownership of any locks it has acquired, subsequently preventing other threads from acquiring them indefinitely. Non-blocking algorithms on the other hand employ mechanisms such as helping or optimistic concurrency control enabling threads to continue even if other threads fail. However, as was hinted earlier, this is only really of relevance for distributed applications that have to deal with the possibility of communication failures. Failures are very unlikely for non-distributed applications.

These arguments seem to imply that non-blocking STMs have tried to provide a one-size-fits-all solution to transactional programming. However, such guarantees are not necessary in general, and as shown in [21], lead to less efficient implementations. In particular, they require indirection, have high logging overheads, require validation, lead to extensively wasted computation and also suffer from the potential for data read to become inconsistent.

Consequently, newer STMs [21, 83, 55, 20] are omitting the non-blocking property, resorting to hybrid blocking/non-blocking or only blocking approaches that significantly reduce these overheads. These new implementations use locks, but whereas traditional ones can block a thread indefinitely thus leading to problems such as deadlock, starvation and priority inversion, these locks can be *revoked* and given to a waiting thread. This means that transactions must still be abortable and thus the overheads of logging writes and the potential for wasted computation are still present. Furthermore, given that the locking policy must be two phase, a problem is introduced for long-running transactions, whereby they may be repeatedly aborted because they hold on to locks past the 'waiting period.' Solutions such as releasing locks early have been proposed but not yet tried [55]. It is interesting to note that using versions for reads and locks for writes, seems to provide better performance than using locks for both reads and writes [83]. This is because of the effects on cache that occur from multiple threads updating the lock value and the expense of upgrading from read locks to write locks.

In comparison, lock inferencing techniques conservatively prevent against deadlock, but given that they use traditional locking, they suffer from the problem of starvation. Furthermore, transactions don't require knowing which objects are accessed at compile time and thus don't suffer from the problem of aliasing and assignments (see Section 2.3), although they do have to enforce

two-phase locking. This is achieved by releasing locks at the end of the transaction [21, 83] or only when required by another transaction (the holding transaction is first given a chance to complete after which it is aborted) [55]. Lock inferencing would avoid upgrading read locks to write locks because of the potential for deadlock, however, the effects on cache coherency of multiple threads updating the read lock is a problem and will need to be taken into consideration.

AtomJava [55] is a particularly interesting state-of-the-art lock-based STM because it is a source-to-source translator for standard Java programs. Before accessing an instance field, the thread acquires a lock on the object. Object locks are implemented as fields that hold a reference to the currently owning thread (`null` indicates that the object is free to be locked). Hence, when a thread locks an object, this `currentHolder` field points to it. When in an atomic block, assigning to a field causes a log entry to be made, consisting of the object reference, the old value and an `UndoObject` with an undo function which reverses the assignment in the event of roll back (this undo code is automatically generated by the translator). If a thread attempts to lock an object that is being held by another thread, it requests the thread to release it as soon as possible and after a number of polite requests, the holding thread is forced to roll-back and the requesting thread is granted access. This provides fair scheduling, ensuring that long transactions don't cause starvation, although one could envision the use of contention managers that determine whether/when a lock can be revoked.

2.2.2.4 Conclusion

Although STM was originally intended as an easy and more efficient way of implementing high-level non-blocking synchronisation operations, many think that it should be provided as a generic abstraction in programming languages (that is, as an implementation for atomic sections). This is because it can afford more parallelism than traditional locks; it doesn't suffer from the problems of deadlock, priority inversion, convoying and starvation; and its ability to roll back can also lead to some desirable abstractions for programmers [44].

However, one significant hurdle it faces is expressiveness, given that atomic sections may contain operations that cannot be reversed. Buffering is one proposed solution [42, 54], although it requires rewriting I/O libraries and is not even applicable in all situations. For example, consider an atomic section that performs a handshake with a remote server. Other implementations forbid irreversible actions using the type system [44], while some throw exceptions [82], although these are not practical in general.

Another major problem is the significant overhead imposed including wasted computation that occurs due to executing transactions destined to abort. A lot of work has been carried out to improve this over the last few years, such as the gradual omission of non-blocking guarantees [21], the introduction of object-based STMs [70] and the ability to customise contention management policies [52, 84]. However, current state-of-the-art lock-based STMs still require roll-back to avoid deadlock and starvation. Consequently, they still incur many unnecessary overheads due to logging, given that the occurrence of deadlock is rare.

This project will employ lock inferencing rather than software transactions, although we hope that this section on transactional memory has given the reader a richer understanding of this competing technique. Furthermore, it also serves to back our choice, given the recent trend of eliminating the non-blocking guarantee: this demonstrates that using locks to implement atomic sections is definitely a step in the right direction.

2.3 Lock inferencing

By far the most popular technique for implementing atomic sections at present is software transactional memory (STM). However, as illustrated in the previous section, it has a number of shortcomings which limit its practicality:

- **Irreversible operations:** Atomic sections implemented using transactions are restricted to operations that are reversible. In [44] this is enforced using the type system, however, this isn't practical in more general languages such as Java. Alternative solutions include buffering [42] and resorting to mutual exclusion locks [92].
- **Performance overhead:** STM incurs significant overheads due to logging, validation and committing. In more recent STMs that use locks [55], there is no need for an explicit validate or commit phase as they acquire ownership of objects before accessing them. Nevertheless, they still have the overheads of logging in case they have to rollback (in order to avoid starvation and deadlock).
- **Wasted computation:** CPU cycles used to execute a transaction that is later aborted is wasted computation. This is inefficient as such CPU time could be used to execute other threads. In one benchmark [55], it was found that tens of roll backs were occurring per second.
- **Need for hardware support:** Due to the performance implications of STMs, it almost necessarily requires hardware support to be practical. However, HTMs are still not quite there yet and face the tough task of convincing chip manufacturers of their usefulness.

These limitations exist because transactional memory (with the exception of [55]) detects interference rather than prevent it. Consequently, it requires that transactions be able to roll back, which has a negative effect on the expressiveness and performance of atomic sections.

Locks overcome these difficulties because they do not allow conflicting accesses to proceed in parallel and thus do not require the need to undo. However, lock-based synchronisation has to be manually enforced by the programmer and is therefore easy to get wrong with the potential for introducing deadlock and even re-introducing races. This has led to a completely different approach to atomic sections that takes a preventative approach by using locks but with little or no effort from the programmer.

Pessimistic atomic sections [69] statically infer the locks that need to be acquired to ensure atomicity and inserts the necessary acquire and release operations. This is different from recent lock-based STMs [55] that also use locks, because pessimistic atomic sections ensure that locks are acquired in a way that prevents deadlock, typically by imposing some ordering as a result of a whole program analysis, whereas lock-based STMs acquire locks as and when they are required (that is, just before accesses occur). Figure 2.5 shows an example of a pessimistic atomic section.

Such an approach, also known as *lock inferencing*, has a number of advantages over TMs, in addition to not suffering from the limitations mentioned above:

- **Better performance in the uncontended case:** A program typically contains some shared objects that will be mostly contended and other shared objects that will be mostly uncontended. The performance overheads of TMs are incurred regardless of whether there is contention or not. Locking on the other hand, can be extremely efficient in the uncontended case, with a

Figure 2.5: *Lock inferencing example that uses reader/writer locks.*

```
void m(Counter c) {
  atomic {
    c.increment();
  }
}
```

(a)

```
void m(Counter c) {
  lockrw(c) {
    c.increment();
  }
}
```

(b)

lot of work having been done in optimisations for it [7, 1]. In some cases, this can be as cheap as setting/clearing a bit [92].

- **Less runtime overhead:** Lock inference techniques may infer the deadlock-free locking policy at compile time and thus the only runtime overheads are the lock/unlock operations. These can be extremely efficient in the uncontended case, as mentioned above. However, even in the contended case, techniques such as *adaptive locking* [34] can be used to reduce the overheads caused by suspending/resuming threads when locks are held for short periods of time.

The magic behind lock inferencing is in the static analysis that determines the locking policy. This analysis has to ensure good performance and freedom from deadlocks; however it must also be *safe*. That is, the locking policy it infers should not lead to errors. The following sections look at issues that must be taken into consideration to ensure the analysis meets these requirements and how existing work in this area has approached them.

2.3.1 Mapping data to locks

Pessimistic atomic sections require that there is a mapping from data to locks, given that they must first acquire the lock for a data item before proceeding to access it. This mapping, also known as the locking granularity, can have a significant impact on the amount of concurrency permitted. For example, if the granularity is coarse, several data items are protected by the same lock; thus preventing concurrent accesses from proceeding in parallel. On the other hand, a finer granularity associates very few data items with each lock, thus reducing the chance of contention and increasing the amount of parallelism possible.

Given that such atomic sections aim to use locks without exposing the programmer to their details, determining this mapping is the responsibility of the static analysis. In approaches where the number of locks is bounded by the size of the program [53], this typically involves associating locks with syntactic entities such as classes. However, given that the size of a program is finite, so is the number of locks. As a result, such approaches don't scale well for programs that consist of a large number of objects at runtime. For example, [53] associates a lock with each point in the program where an object is constructed (e.g. using `new`). While this makes the analysis easier (as locks can be determined at compile-time), it does not scale well because several objects may be constructed using this same code and will consequently share the same lock (see Figure 2.6 for an example).

Ideally, lock inferencing should permit as much concurrency as possible. Therefore, the number of locks should not be bounded by the size of the program, but instead scale with the number of objects, preferably one for each. This is not possible with approaches that infer which memory

Figure 2.6: *Scalability problems when the number of locks is bounded by the size of the program. In [53], locks are associated with object construction code. Consequently, this causes several objects created at the same point in a program to be protected by the same lock and thus concurrent accesses to them cannot proceed in parallel.*

```
class Bank {
  Account newAccount() {
    return new Account();
  }
}

Bank b = new Bank();
Account a1 = b.newAccount();
Account a2 = b.newAccount();
```

```
Thread T1:
  atomic {
    ...access a1...
  }

Thread T2:
  atomic {
    ...access a2...
  }
```

locations are being accessed and subsequently their associated locks (such as [53]) because firstly, the total number of objects that exist during the lifetime of a program can be unbounded and secondly, variables may refer to different objects in different executions of an atomic section. Hence, this would result in a huge number of memory locations being inferred and consequently locked on each execution of the section, even though most would not need to be.

Several approaches support such fine-grained locking, by instead inferring special syntactic expressions that resolve to the accessed memory locations at runtime. For example, [18] infers paths much like those used in Java with the `synchronized` keyword. In this approach, each object is protected by its own lock (again like in Java), thus the object address itself is all that is needed to determine the associated lock. Note that all prefixes of a path must also be locked in order to prevent the object that the path refers to from being changed by another thread and thus meaning that we have acquired the wrong lock.

In another proposal [69], programmers specify which locks protect which variables through guard annotations, thus inferred paths do not refer to objects but instead the locks protecting them. This also has the requirement that all prefixes must be locked. Unlike the object paths approach, objects may be stored in different places from the object, thus suffering some performance penalties due to caching. In summary, these syntactic expressions abstract the *actual* locations being accessed and thus makes supporting finer-grained locking feasible. Figure 2.7 shows an example for each.

2.3.1.1 Read/write locks

The most common type of lock is a mutual exclusion lock (also known as a *mutex*), which ensures that only one thread accesses an object at any one time. However, we can afford even more parallelism by noting that concurrent reads on the same object can proceed without causing interference. This can be achieved using read/write locks. Conceptually, each object has two locks associated with it, one that must be acquired before performing a read, and the other for writes. The read lock can be acquired provided that there are no writers, while the write lock behaves like a mutex. All surveyed approaches do acknowledge this as a future extension (excepting [69] that already supports it), however given that it is a straight forward extension, it isn't discussed in great depth.

To support read/write locks, the static analysis needs to infer the effects [37] of each statement. That is, what reads and writes are performed in it.

Figure 2.7: *Examples of approaches that support fine-grained locking. The right hand column of each shows the transformation that results from their respective static analyses. (a) shows an example of path inference [18] and (b) shows an example of lock inference in the Autolocker tool [69]*

```
class Node {
  Node next;
}

Node n = new Node();
```

```
atomic {
  n.next = new Node();
}

lock(n) {
  n.next = new Node();
}
```

(a)

```
struct Node {
  struct node *next;
};

mutex lock;
struct Node n
      protected_by(lock);
```

```
atomic {
  n.next = malloc(...);
}

begin_atomic();
acquire_lock(&lock);
n.next = malloc(...);
end_atomic();
```

(b)

2.3.1.2 Multi-granularity locking

When traversing dynamic data structures, read/write locks are typically used in a hierarchical manner. For example, in the case of a hash table, a read/write lock will be used to protect its arrays plus individual read/write locks for each bucket; thus looking up an element would involve acquiring the read lock protecting its arrays and then the read lock on the bucket the element's key hashes to. On the other hand, if the hash table is being resized then a write lock would be acquired on its arrays thus preventing the need to acquire locks on the buckets.

Such an organisation, called *multi-granularity* locking in database theory [36, 63], allows more concurrency because concurrent accesses on different buckets can proceed. However, in reality achieving this is not as easy as in databases due to aliasing. For example, another object could potentially hold a reference to one of the hash table's buckets, which could therefore be accessed without going through the hash table and thus circumventing the need to acquire the read/write lock on its arrays. Autolocker [69] overcomes this problem using the notion of *subordinated locks*; that is, a lock L1 declared as subordinated to a reader/writer lock L2 implies that L2 should also be acquired whenever L1 is acquired. If L2 has already been acquired for writes, the acquisition of L1 will be supressed thus providing multi-granularity locking even if objects are accessed through ways they are not supposed to.

This hasn't been explored in other approaches, however, it is likely that it will require programmer annotations like in Autolocker.

2.3.2 Acquiring/releasing locks

Recall that atomic sections ensure that the overall outcome of a concurrent execution is equivalent to one without any interleavings, that is, their execution is serialisable. TMs ensure serialisability by buffering memory updates until the end of the atomic section and then performing them in one atomic step. We can achieve the same effect with pessimistic atomic sections (where updates are made in place) by only releasing acquired locks right at the end.

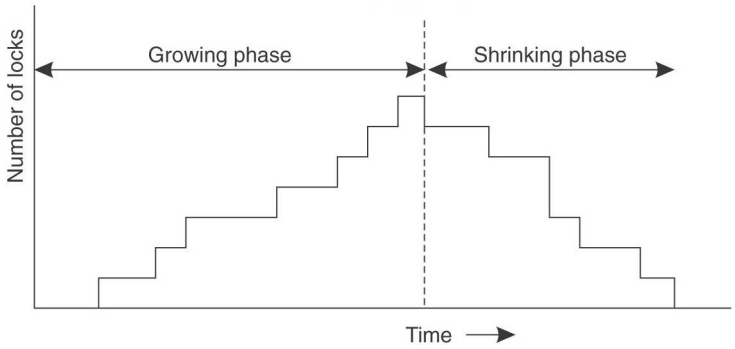
However, in general this isn't strictly required. Database theory says that a locking policy (the strategy used for acquiring/releasing locks) will guarantee serialisability provided that it doesn't acquire any more locks after having released one. This implies that a locking policy more generally consist of a growing phase whereby locks are acquired, followed by a shrinking phase in which locks are released. Such a restriction is known as two-phase locking (2PL). Figure 2.8 is a graphical illustration.

The most basic 2PL policy, also known as conservative 2PL, acquires all required locks at the start of the atomic section and releases them upon having completed executing it. However, this can limit concurrency when objects are only required for a short amount of time and other atomic sections are waiting to access them. Furthermore, large atomic sections may need to wait a long time before they can begin to make any progress. In the worst case this could even lead to starvation, similar to the problem of livelock for large transactions in TMs.

To enable more parallelism, several variations of this basic policy exist:

- **2PL with late locking (strict 2PL):** Delays acquiring a lock until it is absolutely necessary, while unlocking is performed at the end. This has the advantage that atomic sections spend less time waiting before they can start making progress, however, it is complicated by orderings imposed for deadlock avoidance. In the worst case, it can be equivalent to conservative 2PL. This is the locking policy used in [69].

Figure 2.8: *Locking policies that adhere to the two-phase locking (2PL) protocol are guaranteed to be serialisable. It stipulates that a lock must not be acquired after a release operation has been performed, thus such a locking policy in general will consist of a growing phase in which locks are acquired followed by a shrinking phase where locks are released. (Image adapted from http://rainbow.mimuw.edu.pl/SO/Wyklady-html/Tanenbaum/05-26.jpg).*



- **2PL with early unlocking:** Locks are acquired at the beginning of the atomic section, but are released once they are no longer required. This approach can typically achieve more parallelism than strict 2PL [24], as it is not affected by deadlock avoidance (see later), however, it has the disadvantage of requiring to know when objects are no longer required. This is easy when the number of locks is bounded by the size of the program but problematic if not, primarily due to aliasing.

- **Generalised 2PL:** A combination of the above two variations: locks are acquired only when they are needed, and once no more locks need to be acquired, they are released as they are no longer required. Although it can potentially achieve more parallelism than the above two, it is complicated by their respective issues.

2.3.2.1 Which locks to acquire

Pessimistic atomic sections require that objects are locked before they are accessed and thus insert the necessary lock acquisitions at compile time. However, the lock required to protect a particular access is complicated by the following issues:

Assignments

Variables can be re-assigned one or more times throughout the course of the atomic section. As a result, the memory location being referred to by a variable at the point of the access may be different to what it refers to at the point where the lock is being acquired. For example, in Figure 2.9(a), the lock that needs to be acquired to protect the access to `me.account.balance` is the lock protecting `you.account`.

In Autolocker [69] such programs may be rejected if it cannot guarantee that the resulting locking policy will be deadlock free. This may happen if the path for a lock refers to two different concrete locks before and after the assignment. This could potentially occur if the assignment was instead `me = you` and the lock protecting `me.account` was `me.L`. To fix this, their algorithm coarsens the granularity so that the program is then accepted.

Figure 2.9: *Assignments (a) and aliasing (b) affect which locks should be acquired.*

```
atomic {
  me.account = you.account;
  me.account.balance = 0;
}
```

(a)

```
atomic {
  me.account = you.account;
  khilan.account.balance = 0;
}
```

(b)

The approach proposed in [18] infers paths to the objects themselves and acquires locks at the top of the atomic section. Hence, for the access made in the second statement, it pushes the path of the object being accessed, namely `me.account`, through the assignment by rewriting it (although this is complicated by the issue of aliasing (see below)). This paper has the advantage that programs are not rejected.

[25] takes a more conservative approach to dealing with the assignment problem by restricting paths to variables and `final` fields. However, this has the disadvantage of limiting the expressiveness of atomic sections.

Aliasing

Two or more variables may point to the same memory location. As a result, an assignment to an object field accessed through one of the variables may affect which lock to acquire when an access involving the other one occurs. For example, in Figure 2.9, `me` and `khilan` are aliases, thus the object whose balance is being set to 0 is actually that referenced by `you.account` and thus the lock protecting it must be acquired. However, if one is unsure about what may alias what (that is, a good alias analysis is not available), then because the lock inferencing analysis must be safe, all one can do is be conservative and assume that `me`, `you` and `khilan` are all aliases of each other.

The lock inferencing approaches in the literature all have conservative aliasing analyses. For example, Autolocker [69] assumes that all non-global lock paths for the same lock type are aliases, while [18] treats the receiving objects of all paths that have the same final field as aliases. That is, for the paths `x.f.g.s.a.g` and `q.g`, the following are aliases: `x.f`, `x.f.g.s.a` and `q`. Finally, the approach in [53] conflates locks when aliasing makes it ambiguous what objects are being accessed.

Dynamic data structures

Unlike static data structures which have a fixed size known at compile time, dynamic data structures, such as linked lists, can be grown or shrunk as and when needed by the application. This has the advantage that firstly, space is not wasted, and secondly, there is no bound on how big they can be (apart from the limitations of memory). However, this latter characteristic creates a slight complication for pessimistic atomic sections.

Consider the algorithm in Figure 2.10 that traverses a linked list. Given that the analysis is static, we cannot infer (in general) how many nodes will be accessed as we do not know how big the linked list is and thus can only assume that this number is infinite.

In approaches where the number of locks is bounded by the size of the program, this still means acquiring a finite number of locks and therefore is not so much of a problem. However, in fine-grained approaches that associate distinct locks to each object, it is similar to inferring all possible

memory locations being accessed in an atomic section and thus would amount to acquiring an infinite number of locks. This is obviously infeasible and so a better approach is required.

In multi-granularity locking (see earlier), we saw that read/write locks are typically used in a hierarchical way and thus acquiring a write lock on a data structure automatically prevents the need to acquire locks on its internal data structures. In fact, one could say that the write lock *subsumes* its internal locks. However, we also saw that to ensure this hierarchical locking order is adhered to even when aliasing is present, we need to explicitly specify this relationship. For example, Autolocker [69] achieves this using the notion of subordinated locks.

Subsumption can also be used to address the problem of locking an unbounded number of objects by instead acquiring a single lock that subsumes them. This is a generalisation of multi-granularity locking because a lock is acquired on behalf of another lock, instead of being acquired before it. For example, in the case of Figure 2.10, one solution would be to acquire a lock on the `Node` class. This subsumes all nodes being accessed as it prevents other threads from concurrently updating them. However, it also restricts parallelism as it prevents simultaneous traversal of other linked lists and is therefore not desirable.

Another solution is to use the lock associated with the list. This permits different linked lists to be locked independently and thus overcomes the limitation of the previous approach. However, this list lock would have to prevent other threads performing conflicting operations on its nodes. Therefore, like with multi-granularity locking, we would have to specify that the nodes are being protected by the list, also known as a *guarding* relationship [18].

This could be facilitated in Autolocker by allowing objects to be specified as arguments to its `protected_by` annotation. Alternatively, another approach is to use ownership types [11, 18]. In this particular example, one would specify that the main linked list object *owns* the nodes it contains, which is essentially then the guarding relationship.

In papers that infer object paths, a special notation is used to represent paths of unbounded length, otherwise, given that in our example it is not known how many times the while loop will iterate, the analysis would infer an infinite set of paths: {`x, x.next, x.next.next,` ...}. Therefore, they instead represent such infinite sets using a special path. Furthermore, locking such a path has varying semantics. In [15] the notation is `(x.next)*` and locking it amounts to expanding it at runtime until `null` is reached and then subsequently locking each path (in prefix order). This approach has the advantage that it doesn't require guards to be specified, but requires precaution lest there be a cycle. In [18] the path $\texttt{x.next}^{+}$ is used and locking it amounts to acquiring the lock of the guarding object.

2.3.2.2 Acquisition order

So far we have seen that implementations of pessimistic atomic sections have two fundamental requirements, namely that objects must be locked before being accessed and that the locking policy must be two-phased. However, there are certain restrictions on the order in which locks must be acquired in the growing phase, otherwise it is possible for a number of problems to occur:

Deadlock

If two or more threads attempt to acquire the same locks but in different orders, it is possible for them to enter a deadlocked state whereby they wait on each other (see Figure 1.4). This implies that atomic sections should acquire the same locks in the same order, typically achieved by associating a unique number with each lock and then using this to form a total ordering.

Figure 2.10: *When traversing a dynamic data structure, we do not (in general) know at compile time how many objects will be accessed. Therefore, we can only assume that an infinite number of objects will be accessed.*

```
Node x;

atomic {
  x = list.getHead();

  while(x.next)
    x = x.next;
}
```

When the number of locks is bounded by the size of the program [53], it is possible to enforce this ordering at compile time. This is because all locks that need to be acquired in an atomic section are known. On the other hand, with fine-grained approaches that abstract away these details by inferring path expressions, this isn't possible without being overly conservative.

For example, recall that Autolocker [69] permits programmers to specify (through guard annotations) which locks protect which objects, therefore allowing each object to be protected by a different lock. However, to be able to impose an ordering at compile-time, it has to be overly protective by treating all paths for the same lock type as aliases. This has the side-effect that because of certain other dependencies on the locking order due to assignments (see before) and the fact that Autolocker uses late locking, it is more likely that deadlock freedom will not be achievable (without resorting to other means such as coarsening the locking policy).

To obtain an accurate ordering for such approaches, one would ideally need to use the address of the actual object/lock, as this is guaranteed to be unique for all objects [18]. However, this means that the order in which locks are acquired has to be determined at runtime. Furthermore, to facilitate acquiring locks in order, all objects/locks to be acquired must be known at the start of the atomic section (and stored in some collection L).

If late locking is used, as in Autolocker, then just before locking an object for the first time, all objects with addresses lower than it in L would also need to be locked. This amounts to searching L before each lock acquisition for objects with lower addresses, given that paths can be changed by other threads in the mean time. Note also that if the first object accessed in the atomic section has the highest address, all objects will have to be locked before execution can proceed, thus the locking policy would behave in this particular case like conservative 2PL and subsequently afford no additional parallelism.

Conservative 2PL (as used in [18]) and 2PL with early unlocking both acquire all required locks at the start of the atomic section before proceeding. This has the advantage that it doesn't have other dependencies on the locking order due to assignments. The code for deadlock free acquisition at runtime is shown in Figure 2.11. Note that in (a), it is possible for another thread to change the object being referred to by a path during this locking phase; thus a check needs to be made to ensure that the paths still refer to the same objects (b).

One problem with the above is that threads essentially acquire locks in a non-blocking manner, resulting in a lot of unnecessary busy waiting and suspending/resuming when locks are held for long

Figure 2.11: *For lock inferencing approaches that support fine-grained locking and thus infer paths, the actual locks to be acquired cannot be known at compile time. Therefore, obtaining an ordering on the locks is not possible (without being overly conservative) and has to be deferred until runtime. This figure shows example code suggested in [18]. Note that (a) may lead to the wrong locks being acquired if locks are not acquired in prefix order. This may occur because prefixes are not guaranteed to be ordered by address. Hence, after acquiring the necessary locks, a check needs to made to ensure that the paths still point to the same objects, as shown in (b). Furthermore, observe that even if two prefixes have their addresses swapped (due to thread interference), this won't matter because locks will still have been acquired on those objects. This is because we are only using mutual exclusion locks here. However, if using reader/writer locks we would need to check that each individual path is pointing to the same object.*

```
Object a[n] = {path_1 ... path_n};
sort(a);
lock(a[0]) ; ... ; lock(a[n]);
```

(a)

```
boolean locked = false;

while(!locked) {

  // lock in address order
  Object a[n] = {path_1 ... path_n};
  sort(a)
  lock(a[0]) ; ... ; lock(a[n]);

  // check that paths still point to the same objects
  Object a_after[n] = {path_1 ... path_n};
  if(!a.equals(a_after))
    unlock(a[0]) ; ... ; unlock(a[n]);
  else
    locked = true;

}
```

(b)

periods of time. One radical alternative, not yet considered in the literature, is to integrate this process with the scheduler. That is, a thread passes a set of paths that it wishes to lock atomically to the scheduler and then blocks. The scheduler acquires these locks on its behalf when they are all available (while ensuring fairness) and subsequently resumes the thread. This has the advantage that CPU time is not wasted, although efficiently implementing such atomic acquisition of a set of locks could prove to be a significant challenge.

Alternative approaches for deadlock freedom include using the type system [11].

Wrong locks being acquired

Current proposals for supporting an unbounded number of objects infer special path expressions that refer to either the object to be locked or the lock that protects it; an example path is `x.f.g`. It was noted that in order to prevent the object/lock that this path points to from being changed by another thread, all prefixes of the path (`x` and `x.f`) should also be locked. However, it is still possible for interference to occur if the prefixes are not acquired in the right order, namely in increasing prefix length order. Thus, for `x.f.g`, the correct order for locking is `x`, `x.f`, `x.f.g` [18].

However, this introduces the possibility of deadlock, as locks are not acquired in accordance with some global ordering. We can overcome this by detecting when deadlock occurs, unlock all locks already acquired and retry, although this is only possible if all locks are acquired at the top of the atomic section otherwise we would need to rollback writes. Nevertheless, it has the advantage that runtime costs are lower as no searching/sorting is required [18].

2.3.3 Minimising the number of locks

So far it has been assumed that all object accesses should not proceed without first acquiring its protecting lock. This is a fundamental requirement for pessimistic atomic sections, as it ensures that other threads cannot perform conflicting updates at the same time. However, it is often the case that only a subset of these locks need to be acquired to ensure atomic execution. For example, only shared objects need to be locked because only they can be accessed by multiple threads.

Minimising locks in this way has the advantage of reducing the runtime overheads imposed by acquiring/releasing locks. However, how much it can actually reduce this depends on whether it is performed statically or dynamically. Static analyses [17, 53, 54, 15] have to consider all possible executions of the atomic section and so will be less optimal than dynamic analyses that can make decisions based on current thread behaviour [78]. For example, in approaches that infer paths, if a path can refer to both shared and local objects, then it cannot be removed if the analysis is static, whereas a dynamic analysis would treat each binding separately. On the other hand, dynamic analyses need to speculate thus requiring the ability to roll back if they make a mistake, whereas static analyses are safe. Furthermore, dynamic analyses will impose their own runtime overheads, which will most probably render them useless.

In this section, we look at when a lock may be unnecessary and the associated (static) techniques for detecting such cases.

2.3.3.1 Thread shared vs. thread local

Recall that interference occurs when two or more threads simultaneously perform conflicting operations on an object. However, if an object is only accessed by one thread, that is, it is *thread local*, then it is not possible for interference to happen and thus a lock does not need to be acquired for it. A number of techniques exist to determine which objects are thread shared and which are thread

local:

Escape analysis

This is a whole-program analysis whose aim is to determine if an object is accessed outside the method and/or thread it was created in. Central to it is the notion of *escapement* [17]:

- An object O escapes a method M if the lifetime of O may exceed the lifetime of M. This allows objects to be allocated on the stack avoiding the overheads of garbage collection.
- An object O escapes a thread T if another thread $T' \neq T$, may access O.

Furthermore,

- If an object O does not escape a method M, then this implies that it also doesn't escape the thread T in which M was invoked/O was created in.

The analysis [17] first performs an intraprocedural analysis for each method, building up a connection graph. Nodes in this graph represent reference variables and objects (constructed using `new` like in [53]) while edges describe the connection between variables and objects. These edges are of two types: *points-to edges* between a variable node and an object, and *deferred edges* between two variable nodes. The latter type is used to model assignments that merely copy references from one variable to another [17]. During graph construction, an object can be classified as *NoEscape* (local to method), *ArgEscape* (escapes the method via its arguments but does not escape the thread) or *GlobalEscape* (escapes threads and methods). This is updated iteratively using the following observations:

- An object reachable from a *global variable* node is classified as GlobalEscape.
- An object reachable from an *actual argument* node (as opposed to from a *formal argument* node) is classified as ArgEscape
- Objects reachable from other objects that are classified as GlobalEscape are also classified as GlobalEscape.
- Objects reachable from objects that are classified as ArgEscape are also classified as ArgEscape (unless they are already GlobalEscape).

After the intraprocedural stage is complete, an interprocedural analysis in conjunction with a *program call graph* (that describes method calls) is performed to combine the connection graph of callees with that of callers. Finally, thread local objects are those which are classified as NoEscape or ArgEscape. Please refer to [17] for the full details.

Continuation effects

The effect of a statement s is the set of locations that may be dereferenced or assigned to while executing it. Thus, continuation effects are the locations that may be accessed during and/or after the execution of s [53]. They can be divided into *input effects* denoting the locations that may be accessed during and/or after s, and *output effects* for those accessed after.

Most multi-threaded programming languages provide mechanisms for spawning threads, such as `fork()` in C and `Thread.start()` in Java. The significance of continuation effects is that by

Figure 2.12: *Code for continuation effects example.*

```
Counter c1 = new Counter();
Counter c2 = new Counter();

spawn {
  for(int i=0; i<99; i++)
    c1.increment();

  c2.increment();
}

c1.increment();
```

calculating the input effect of the spawned child thread, namely the locations accessed by it, and the output effect after the thread creation call in the parent thread, and then taking the conjunction, we can infer those locations that may be accessed by both child and parent threads.

Figure 2.12 shows an example to illustrate this. The example uses a `spawn` construct that executes the containing code in a separate thread. The spawned thread accesses the `counter` values of objects `c1` and `c2`, and thus its input effects are `c1.counter`, `c2.counter`}. Note that as we are interested in the input effects of the *spawned thread*, as opposed to the *spawn statement*, this means that it only includes locations accessed by the thread itself.

The output effect of the `spawn` are those locations accessed after it in the parent thread, namely `c1.counter`. Hence the intersection gives us `c1.counter` and thus we infer that only `c1` is shared. Note that even though `c2` is created in the parent thread, it is only accessed in the child thread and is thus considered thread local to it.

This example didn't include atomic sections as it is just illustrating the concept of continuation effects. However, one could imagine extending it so that the effects of atomic sections are considered. For example, consider the child's thread and `c1.increment()` in the parent thread being wrapped inside atomic sections. Then, in both atomic sections `c1` has to be locked due to the reasons mentioned above.

2.3.3.2 Conflating locks

In [53], it is observed that if in every atomic section that accesses ρ', ρ is also accessed, then only ρ needs to be locked and not ρ'. The paper describes this as ρ dominating ρ'.

This optimisation removes redundant locks but is only applicable when the number of locks is bounded by the size of the program, given that the *actual* locks that will be acquired are not known when they are unbounded. Furthermore, it doesn't always lead to the minimal set of locks when no dominator exists, even though *it is* possible to reduce the number of locks. Figure 2.13 demonstrates this point.

Figure 2.13: *The 'dominates' algorithm of [53] does not always lead to the minimal set of locks required as demonstrated by this example. Three threads are each executing distinct atomic sections that access locations shared with the other two threads (a, b and c are memory locations). If the locking policy is conservative 2PL, neither atomic section can be executed simultaneously, as it would need to wait until the other two threads have finished executing. Hence, only one thread can execute at a time and thus only one lock is required for all three atomic sections. However, the 'dominates' algorithm relies on dominating memory locations, which do not exist. As a result, it cannot even reduce the number of locks let alone infer that only one is required.*

Thread 1	Thread 2	Thread 3

`atomic {`
`  ...access a...`
`  ...access b...`
`}`
	`atomic {`
	`  ...access b...`
	`  ...access c...`
	`}`
		`atomic {`
		`  ...access c...`
		`  ...access a...`
		`}`

2.3.3.3 Constant paths

If a path is constant, it cannot be modified by another thread and thus does not need to be locked. Although languages like Java, in which constant paths are denoted using the modifier `final`, allow deferring the initialisation, which incidentally could occur in an atomic section. If this is the case, then this optimisation cannot be used.

2.3.3.4 Inside atomic sections vs. outside atomic sections

In addition to the distinction between thread shared and thread local data, as well as data that is constant, [54] makes the novel proposal of also distinguishing between objects that are accessed inside atomic sections and those not. This is used to remove unnecessary read/write barriers outside atomic sections that are inserted to ensure strong atomicity in their source-to-source translator. However, with regards to path inference, it could potentially be used to filter those paths which may refer to thread-shared objects but which are only accessed outside atomic sections.

2.3.4 Starvation

In [43], the problem of non-terminating atomic sections in STMs is discussed. While the particular scenario in the paper (reproduced in Figure 2.14) cannot occur with lock inferencing techniques, the potential for non-terminating atomic sections still remains. This could be a serious problem because the executing thread will hold on to acquired locks starving others indefinitely. Incidentally, starvation can also occur when a thread blocks while holding a lock, although it is assumed that it will eventually be resumed and thus release the lock.

TMs don't have this issue because transactions can be rolled back, however, pessimistic atomic sections don't have this luxury and thus this problem cannot be prevented completely. However, it can be minimised by locking policy optimisations such as early unlocking, late locking and downgrading write locks to read locks (note: upgrading of read locks to write locks can lead to deadlock [80]).

No lock inferencing technique has looked at addressing this issue directly yet, perhaps because it isn't possible to overcome.

Figure 2.14: *Starvation due to non-terminating atomic sections. This example (although not originally object oriented) was presented in [43] to demonstrate how conflicting updates made by other threads can lead to non-terminating transactions (in STMs). In particular, if thread T1 reads the value of* $x.val$ *before thread T2 executes (thus having value 0) and then reads* $y.val$ *after T2 commits (subsequently having value 1), the* if *condition evaluates to true and the infinite loop is executed. This does not occur with lock inferencing techniques as* x *and* y *are locked before being accessed. However, it doesn't eliminate the possibility of atomic sections that do not terminate. This is a problem for lock inferencing techniques because it prevents other threads from ever acquiring the necessary locks. For example, if* $x.val$ *and* $y.val$ *were originally different, it would cause T1 to enter the infinite loop hence preventing T2 from ever proceeding (assuming that T1 acquired the locks first).*

```
class Integer {
  int val = 0;

  Integer(int initial) {
    val = initial;
  }
}

Integer x = new Integer(0);
Integer y = new Integer(0);
```

```
Thread T1:
  atomic {
    if(x.val != y.val)
      while(true) { }
  }

Thread T2:
  atomic {
    x.val++;
    y.val++;
  }
```

2.3.5 Nested atomic sections

Several different semantics have been proposed for nested atomic sections in transactional memory systems [72] to minimise its overheads and also allow inter-thread communication between atomic sections. However, with lock inferencing it hasn't been looked at in any detail. This may be because of the requirement for the inferred locking policy to be two-phased and/or locks requiring to be acquired in some order to ensure that deadlock is avoided. Hence, nested atomic sections are currently just merged with their parent.

However, this poses concerns for the liveness properties of programs, especially when paths are locked at the top of the atomic section, given that there may be a large number of them.

2.3.6 Source code availability

Lock inferencing performs a whole-program analysis to infer which locks need to be acquired. However, current techniques require the source code being available. This can be a problem when using external libraries as these are normally provided in a compiled form. In this case, a byte-code analysis would be required instead, although this is orthogonal to this project given that we are interested in the more fundamental challenges that determine whether pessimistic atomic sections are viable or not.

2.3.7 Conclusion

STMs have generated intense interest over the last few years and are currently the most popular technique for implementing atomic sections. However, they still have a number of significant hurdles that they need to overcome, most critical being their inability to flexibly support irreversible opera-

tions. Furthermore, the overheads incurred by STMs is insensitive to the amount of contention there is, thus leading to significantly slower execution even in the common case of uncontended execution. Hence, this has led to the consideration of a different approach to atomic sections, namely that of lock inferencing. This technique statically infers the locks that need to be acquired to ensure atomicity and inserts the necessary acquire and release operations. This is different from recent lock-based STMs because such pessimistic atomic sections ensure that locks are acquired in a way that prevents deadlock, thus not requiring the need to rollback.

The magic behind lock inferencing is in the static analysis that determines the locking policy. This analysis has to ensure good performance and freedom from deadlocks, however, it must also be safe. Performance is determined by the amount of parallelism that the locking policy can afford, which depends on the granularity of the locking as well as the type of locks used. In some papers [53], the number of locks is bounded by the size of the program resulting in a coarse granularity that doesn't perform well in programs that have a large number of objects. Other approaches [69, 18] support fine-grained locking, with each object having its own lock. This affords more parallelism as it allows concurrent accesses of different objects.

However, this poses a challenge when determining which locks to acquire in an atomic section, given that in such approaches the number of locks may be unbounded (as the number of objects can be unbounded). Existing work has dealt with this problem by inferring special syntactic expressions that resolve at runtime to the object/lock being accessed. This has the advantage over approaches that infer and lock all possible memory locations which may be accessed in an atomic section, in that they are significantly smaller in number and also result in only locking the actual objects being accessed at runtime.

Performance is also dependent on the type of locking policy used. Database theory says that locking policies should be two phased (2PL) to ensure that the resulting concurrent interleaving can be made equivalent to one in which atomic sections are executed one after the other, or alternatively, to ensure that atomic sections are serialisable and thus ensure atomicity. A two phased policy stipulates that additional locks must not be acquired after a lock is released implying that there be a growing phase during which locks are acquired followed by a shrinking phase during which locks are released. A simple version of this would be to acquire all locks at the beginning of the atomic section and release them at the end. However, to permit as much concurrency as possible, locks should be held for the shortest period of time. Thus, a number of variants of the aforementioned basic policy exist such as late locking (strict 2PL) that acquires locks only when absolutely required and early unlocking that releases locks as soon as they are no longer needed. Additional optimisations include using reader/writer locks and minimising the number of locks using techniques such as escape analysis and continuation effects.

Pessimistic atomic sections are further complicated by the need for the locking policy to avoid deadlocks. This typically requires ensuring that locks are acquired in some globally defined order. In approaches where the number of locks is bounded, this can be determined statically, while in dynamic approaches that infer path expressions, it is not possible without being unduly conservative. To overcome this, ordering should be performed at runtime, although this has the disadvantage of imposing additional overheads such as for sorting.

Furthermore, paths must be locked in prefix order to prevent the wrong lock being acquired. As a result, the aforementioned technique has the additional overhead of having to check after acquiring all the locks that the paths still refer to the same memory locations. Note that the occurrence of deadlock is rare, hence instead of actively avoiding deadlock we can instead let it occur and then deal with it when it occurs (by releasing acquired locks and retrying to obtain them). Although this

has the requirement that all locks must be acquired together (otherwise we would have to rollback writes) and thus means that late locking (strict 2PL) cannot be used.

Additional complications to the above issues include assignments, aliasing and the problem of traversing dynamic data structures whereby one cannot in general know at compile time how many elements they will contain and thus how many objects will be accessed. As a result, one has to assume that a potentially unbounded number of elements may be accessed. Solutions include using guards [69, 18].

Existing work has only looked at a subset of features that modern programming languages provide. In particular, features such as subclassing, polymorphism, arrays, exceptions, recursion and inter-process communication still haven't been addressed yet, while alias analyses can be overly conservative at times. This project looks at extending existing work by considering such features. It will be a direct extension of the work done on path inference [18].

There are also a number of other more worrying problems related to progress such as starvation, which will also be looked at in this project.

2.4 Hybrids

One of the biggest problems with transactional memory is the overhead incurred for supporting roll back. This overhead is insensitive to the amount of contention there is and can lead to significantly slower executions in the common case of when there is none. Furthermore, they have trouble dealing with irreversible operations, having to revert to mechanisms such as buffering, which incidentally do not provide a general enough solution. However, they do have the advantage that they can afford more concurrency. Locks on the other hand have the advantage of being extremely efficient in the uncontended case as well as not suffering from problems of expressiveness, although in general they don't permit as much concurrency, especially if mutual exclusion locks are used.

As a result, a hybrid approach has been considered [92] (in the context of Java) that uses locks when there is little or no contention or if an irreversible operation is encountered and software transactions (object-based STM) otherwise. Recall that programmer intent is mostly atomicity when using mutual exclusion locks, hence this paper considers `synchronized` blocks (that protect the same object) as atomic sections (referred to as monitors in the paper).[3]

Contention occurs when a thread tries to acquire a lock already held by another thread, that is, two threads are attempting to execute a `synchronized` block protecting the same object. If this happens, execution switches to using transactions, although only after the lock is released by the currently holding thread. Furthermore, if a thread is currently executing a monitor transactionally and encounters an irreversible operation, such as I/O, it reverts to using locks. For this, a log must be kept of all locks that would have been acquired if the outermost monitor had executed using locks. If a lock cannot be acquired because it is being held by some other thread, the innermost monitor (of the current thread) is rolled-back and re-executed.

In theory this approach should provide efficient execution when there is no contention and scale fairly well when there is, however, it has a number of problems. Firstly, while atomicity is what `synchronized` blocks are commonly used for, it is not always the case. Moreover, even if a programmer had intended on atomicity, what is to say that they have used synchronisation correctly? Additional checks would still need to be made to verify this, using type systems for example. This implies that such an approach does not provide an abstraction as such but is instead aimed at

[3] Although it has been shown that executing `synchronized` blocks as transactions may break a program that relies on races for progress [10].

improving performance. Furthermore, the paper does not detail how deadlock is avoided, given that a wait-cycle may occur due to threads having to wait for locks to be released before executing the monitor transactionally. Finally, performance comparisons using mutual exclusion locks are not very revealing given that reader/writer locks afford much more parallelism, although in the context of Java, mutual exclusion locks are appropriate.

2.4.1 Conclusion

Hybrid execution is an interesting idea and one could envision using lock inferencing to infer which locks need to be acquired for atomic execution and then applying the optimisation proposed in this paper, although significant overheads occur to support this approach as shown in the paper's benchmarks. For example, in the uncontended case, these can be upto 50%! One thing we can take from this paper is that it once again gives promising evidence that any implementation for atomic sections will need to use locks in some way or another.

Chapter 3

Specification

When trying out new language features, language designers take one of three approaches:

Modifying an existing language implementation

An existing language, such as Java, is modified to support the feature natively by making amendments to the compiler and/or runtime system [43, 82]. This can be a significant engineering challenge, given that these systems are typically huge and complicated, with lots of subtle details that need to be taken into consideration. Furthermore, modifications of this type require patching the runtime/compiler, which may not be welcomed by the vast majority. It also quickly becomes obsolete if not kept up-to-date with newer versions of the language implementation. Thus, this option should only be chosen when one is contemplating on actually adding the feature to the language, with most conceptual hurdles having been overcome.

Lock inferencing is still at a stage where there are a number of fundamental areas and issues that need to be looked at and thus requires a lot of freedom. Furthermore, modifying a language requires spending a considerable amount of time understanding an existing system, which leaves little time for experimentation. Hence, this project will not take this route.

Pre-processing

This approach involves extending an existing grammar to allow programs with the proposed language feature, and then providing a source-to-source translator that compiles these programs into the original language, using its existing language features to implement the proposed one. For example, [55] takes Java programs containing `atomic { }` sections as input, replacing them with the necessary Java code for ensuring atomicity.

This approach has the advantage that it avoids the complexities of modifying an existing compiler/runtime and thus affords fast prototyping. It is easy to distribute and only needs revising when the language itself changes, which is generally a lot less frequent than updates to its implementation. However, it has the disadvantages of relying on an uncontrollable back-end and having to compromise semantics when features are simply unavailable (e.g. rolling back class loading in [55]). Furthermore, it also has the burden of dealing with the full set of features in the existing language. Hence, this project will not take this route either.

Toy language

A simple language is implemented with just those features necessary to evaluate the viability of the proposed language feature. It provides a testbed for flexible experimentation without being engulfed

in implementation detail or a plethora of language features.

Performance is not of primary importance and thus an interpreter may be used for executing its programs. This gives full control over what happens at runtime and also permits close monitoring. Writing an interpreter for an existing language is infeasible given the number and complexity of the features.

However, this approach has the disadvantage that it is essentially throw-away code. That is, it has no use outside of research. Furthermore, given that there are minimal language features, one may have to be creative when coming up with example programs.

Nevertheless, because this approach best facilitates the aims of this project, namely to explore a much wider set of language features than have been covered in previous work, it will be the chosen option. Even though it will not be usable in the practical sense, we hope that the ideas that come about as a result of such flexibility can be built upon by later work to consider extending a real language.

3.1 Language

The aim of this project is to look at a much wider variety of language features than has been considered by any existing work on lock inferencing that supports an unbounded number of locks. However, given the research nature of this project one should bear in mind that it is very difficult to gage how many features will actually be covered.

- `Singlestep` will be object-oriented, although it does not have to be pure. That is, it may have *primitive types* like in Java. However, these may be later represented as objects instead if deemed necessary, such as for allowing a much finer granularity of locking.
- It must be multi-threaded, with mechanisms that allow the programmer to explicitly spawn threads, such as `fork()` in C.
- Each object should have an associated reader/writer lock and the language should provide constructs to lock (and unlock) an object either in read mode or write mode.
- The language should allow programmers to denote that a method/block of code should execute atomically, using the `atomic` keyword for example. The semantics are not provided by the runtime, but by a static analysis that inserts the necessary locks, although runtime support (e.g. scheduler) may be utilised if necessary.
- Given that speed is not of importance and that we are experimenting with a toy language, an interpreter can be used rather than a compiler. This will also make it easier to analyse and control execution behaviour.
- It should support inheritance and polymorphism.
- It should also support arrays (at least one dimensional).
- The language should be statically type checked.
- The language should allow limited I/O such as printing to the screen.
- The language may later support encapsulation, although this is not very important as it would be removed by the analysis anyway to support locking objects at the start of an atomic section.
- It should support exceptions much like Java's `try/catch` block and `throw` statement.

3.2 Analysis

- The main outcome of this project is a static analysis that takes source code written in `singlestep`, and inserts the necessary locks to ensure atomicity.
- The analysis only needs to ensure weak atomicity, that is, atomicity between atomic sections. This is because programs may rely on races for progress, such as in lock-free algorithms. However, an existing race-detection tool could be used to warn the programmer of races (after having inserted locks).
- The analysis will primarily build upon the work done by Dave Cunningham [18] but extended to cover a richer set of language features. Thus, it will infer and lock paths.
- Nested atomic sections should be supported.
- In atomic sections, all objects must be locked before being accessed. The analysis should support fine-grained locking, with each object having its own lock. It should also discriminate between reads and writes, allowing several threads to perform reads concurrently, while resorting to mutual exclusion for writes.
- The number of locks should scale with the number of objects and not be bounded by the size of the program.
- The locking policy inferred should be two-phased (2PL). That is, no further locks should be acquired after a lock has been released. This is to ensure that the resulting concurrent interleaving can be made equivalent to one in which each atomic section is executed one after the other.
- Locks should be held for as short a time as possible. Thus, a variant of 2PL such as late locking or early unlocking should be used. Write locks should also be downgraded to read locks when an object is only read after a particular program point, although read locks should not be upgraded to write locks as this can lead to deadlock.
- It should try to minimise the number of locks by discriminating between thread local and thread shared objects. This can be fine-tuned by additionally distinguishing between objects accessed inside atomic sections and those not.
- It is also hoped that the alias analysis is less conservative than previous approaches. This should be possible when considering the program as a whole as opposed to dealing with each atomic section in isolation.
- Deadlock should not be allowed to occur. This can be achieved by imposing an ordering on lock acquisitions, although given that the number of locks is unbounded, this would have to be done at runtime. Alternatively, paths could be locked in prefix order with deadlock being detected instead of avoided. In this latter scenario, an exception may be thrown and locks re-acquired. This approach has the advantage that it doesn't require sorting of addresses at runtime and is more efficient given the rarity of deadlock occurrence.
- Both dynamic approaches for deadlock avoidance require that all locks are acquired at the top of the atomic section and thus our analysis can only use conservative or early unlocking 2PL.
- The analysis can assume that all source code is available, although in reality it may not be. In the latter case, a byte-code analysis would be required although this is beyond the scope of the project.

- When faced with the task of locking a potentially unbounded number of objects, there are a number of options such as locking the objects' class or using guards. The problem with guards is that they require programmer annotations, which may impose a big overhead for large programs. Hence, a hybrid approach could be used instead where the class is locked by default, but which allows the programmer to improve the precision by providing guarding annotations. Checking whether the guard annotations are correct is beyond the scope of this project.

- Multi-granularity locking refers to the hierarchical use of reader/writer locks, such that acquiring a write lock at one level prevents the need to acquire locks on lower levels. It has the advantage of reducing the number of locks that need to be acquired, although its use is restricted to situations where an object owns other objects and where all accesses must go through that object. Hence, this is not required at all times and is thus not of utmost importance.

3.3 Runtime

- The runtime will not be concerned with speed, although it may wish to apply certain optimisations that are not related to how fast it can execute statements, such as using adaptive locking to prevent threads from being suspended/resumed when locks are held for short periods of time (and the conflicting threads are running on separate cores).

- When executing statements, the interpreter should use small-step semantics to mimic what really happens when code is executed at the machine level and to allow interference to occur.

- Multithreading should be simulated by the runtime as opposed to forking native threads making it possible to closely monitor the execution. This will require a scheduler.

- The runtime should also be able to utilise multiple cores so that the effects of true parallelism can be investigated.

- It should offer debugging facilities in which the state of individual threads can be inspected.

- This debugger should support single stepping.

3.4 Testing

The most important component of this project is the static analysis that infers which paths need to be locked for atomicity. Programmers assume that it will provide this guarantee with errors possibly leading to disastrous consequences [64]. Hence, it should be *critically* tested for correctness. However, given that there are infinitely many inputs, ideally a formal abstraction like that used in Autolocker [69] should be utilised. In this section, we describe example input programs together with the expected output. For brevity, code snippets are small and avoid method calls unless where absolutely necessary. Furthermore, they don't include the (boilerplate) code for deadlock avoidance.

The basic analysis should lock all accessed paths, including all prefixes, at the top of the atomic section. They should be unlocked in reverse prefix order by the end of the atomic section (exactly where depends on whether conservative or early unlocking 2PL is used). It should correctly distinguish between read and write accesses.

Input

```
atomic {
  Object o = x.f.g;
  y.i = 3;
  ...
}
```

Output

```
lockr(x); lockr(x.f); lockrw(y); {
  Object o = x.f.g;
  y.i = 3;
  ...
}
unlock(y); unlock(x.f); unlock(x);
```

It should deal with assignments like in [18]. Write accesses should subsume read accesses.

Input

```
atomic {
  x.acc = y.acc;
  x.acc.bal = 10;
  y.acc = 10;
}
```

Output

```
lockrw(x); lockrw(y); lockrw(y.acc); {
  x.acc = y.acc;
  x.acc.bal = 10;
  y.acc = 10;
}
unlock(y.acc); unlock(y); unlock(x);
```

It should downgrade write locks to read locks when it is sure that the corresponding path is no longer modified after a particular program point.

Input

```
atomic {
  x.f = 3;
  int y = x.g;
  ...use y...
}
```

Output

```
lockrw(x); {
  x.f = 3;
  downlock(x);
  int y = x.g;
  ...use y...
}
unlock(x);
```

When it is possible that an unbounded number of paths may be accessed, the special infinite path [18, 15] should be inferred.

Input

```
atomic {
  while(x.next)
    x = x.next;
}
```

Output

```
lockr(x{.next}*); {
  while(x.next)
    x = x.next;
}
unlock(x{.next}*);
```

If polymorphism is supported by the language, it is possible that some inferred paths will not be valid. Therefore, the analysis should use static type information and discard those which aren't. For example:

Supporting classes and shared objects

```
class Integer { int val; }

class A {
  Integer x;

  void f() {
    print x.val;
  }
}

A a = new A();
B b = new B();
```

```
class B extends A {
  Integer y;

  void f() {
    print y.val;
  }
}
```

Input

```
atomic {
  if(condition)
    a = (A)b;

  a.f();
}
```

Output

```
lockr(a); lockr(a.x);
lockr(b); lockr(b.y);
{
  if(condition)
    a = (A)b;

  a.f();
}
unlock(b.y); unlock(b);
unlock(a.x); unlock(a);
```

The analysis should be safe with respect to aliases. That is, if it is not sure whether two paths may alias each other, it should assume that they do (although considering the program as a whole should improve precision).

Input

```
atomic {
  x.f = y.f;
  z.f.g = 10;
}
```

Output

```
lockrw(x);
lockr(y); lockrw(y.f);
lockr(z); lockrw(z.f);
{
  x.f = y.f;
  z.f.g = 10;
}
unlock(x);
unlock(y.f); unlock(y);
unlock(z.f); unlock(z);
```

3.4.1 Additional tests

A large suite of tests should also be developed to ensure that the various other components work correctly:

3.4.1.1 Language behaviour

- Unit tests should exist for each language feature.

3.4.1.2 Runtime behaviour

- Interleaving of threads should be consistent with the scheduling policy.
- At each step, execution performs a single reduction.
- The debugger should correctly reflect the current state of the runtime.

3.5 Documentation

Documentation is extremely important in any project, as it enables the work carried out to be built upon. As part of this project, a report should be produced detailing decisions taken together with justifications.

Chapter 4

Evaluation

Atomic sections guarantee that the contained code will execute as if in 'one step.' Therefore, an important part of the evaluation of this project is to ensure that the inferred locking policy actually does ensure atomicity. Furthermore, this project's main goal is to consider a much wider set of language features than has been covered by previous work. Hence, we also evaluate our proposed solutions to these by considering a wide range of concurrent problems that capture fundamental aspects of concurrent software. These will also serve the purpose of demonstrating what lock inferencing can and can't do, although at this stage it is hard to tell. Finally, atomic sections should aim to permit as much parallelism as possible. Therefore, we also evaluate performance.

4.1 Verifying correctness

Atomic sections that don't ensure atomicity are useless, thus we must verify correctness for our implementation. Recall that much work has been done to verify atomicity with the most popular approach being the use of type systems with Lipton's reduction [65]. To verify correctness, we will also use this technique.

4.2 Language features

As previously mentioned, the primary goal of this project is to apply lock inferencing to a much larger set of language features than has been considered in previous work. An ideal way of evaluating this would be to apply our implementation to a wide range of concurrent problems. These will also serve the purpose of demonstrating what pessimistic atomic sections can and can't do, although at this stage it isn't clear which category they will fall into.

- **Red-black tree [95]:** This is a tree data structure frequently referred to in the literature for atomic sections/software transactional memory because of the apparent complexity involved in implementing synchronisation correctly for it. We will be the first to demonstrate lock inferencing on it. Additionally, this will enable us to evaluate the ability to cope with potentially accessing an unbounded number of elements.
- **Hashtable:** Hashtables are used in the literature to show how well the proposed approach deals with varying levels of contention as well as its scalability to increasing numbers of CPUs. We will also consider it for this purpose.
- **Bounded buffer/Producer and consumer problem:** This forms a very important class of

programs, whereby producer threads put data into a bounded buffer, which is subsequently removed by consumer threads. One would imagine that the act of putting/taking an element from the buffer would be atomic, however this poses a challenge for atomic sections because it may not be possible for a producer or consumer to continue as the buffer may be full or empty respectively. Recent proposals for STMs include a `retry` keyword [44] that has the effect of rolling back the transaction and blocking it until one of the locations the current transaction has previously read is updated. Hence, if a producer finds that the buffer is full, it will execute `retry`, blocking until the state of the buffer changes. With pessimistic atomic sections, we have to resort to other means given that we cannot roll back nor block while holding a lock on the buffer because this will prevent other threads from accessing it.

- **Dining philosophers problem:** The well known dining philosophers problem is the classic example used for demonstrating the concept of deadlock: A number of philosophers sit around a dining table, each sharing a fork with their neighbour. Each philosopher executes a loop of thinking for a little while, before having something to eat and then thinking again. Before eating, they must acquire both forks, blocking if any of them are currently in use. However, care has to be taken to ensure that deadlock does not occur where each philosopher waits on its successor. With atomic sections, this is really easy to implement given that all it requires is wrapping the eating code inside `atomic { }`. However, this example can generate a considerable amount of contention if philosophers spend very little time thinking, potentially leading to never ending rollbacks in an STM implementation. This is not a problem with pessimistic atomic sections of course and demonstrates one of its strong points.

- **Parallel mergesort:** Mergesort is one of several well-known sorting algorithms in Computer Science. It works by first dividing the dataset into two halves, sorting each half and then merging them together by making pairwise comparisons of elements from each. This is an interesting problem for atomic sections, because although each half can be sorted in parallel, the synchronisation may not allow this given that locking policies must be two-phase and read locks cannot be upgraded to write locks (to avoid deadlock). Parallel mergesort is also a problem for STMs, given that it requires threads to block until both halves have been sorted before performing the merge.

4.3 Performance

Performance is an important factor for any implementation of atomic sections, given that programmers typically want their software to be as fast as possible. However, while ideally we would like to compare our implementation against software transactional memory as well as other attempts at pessimistic atomic sections, this would be beyond the scope of the project. Furthermore, our use of a toy language means that we must abstract away the notion of time as it is meaningless in this context.

4.3.1 Benchmarks

To evaluate the performance aspect of this project, we will benchmark three versions of the aforementioned concurrent problems:

- **Manual locking:** This will include both coarse and fine-grained policies.

- **Lock inference:** Implementation using our atomic sections.

- **Non-blocking:** Implementation that uses non-blocking primitives such as Compare-and-Swap (CAS).

4.3.1.1 Metrics

As mentioned above, employing a toy language means that we cannot reliably use metrics that have a notion of time associated with them and must instead use more abstract ones. However, we do have the advantage of being able to control fully how threads are scheduled and thus can ensure that these are reliable. This project will use the following metrics:

- Number of computation steps taken by a thread to complete an operation (such as inserting an element into a hashtable).
- Number of times that threads are resumed/suspended as a result of lock contention and for how long (in terms of computation steps). This will reveal information about the overheads caused by locking.
- Number of threads (logically) executing in parallel at the same time (or as a proportion of the total number of threads in the system). This will reveal information about the amount of concurrency in the system.

In order to evaluate this project, the language may require additional constructs to those already suggested, although this cannot be determined at this stage.

Bibliography

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 207–222, 1999.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] E. Allen, D. Chase, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr, and S. Tobin-Hochstadt. The fortress language specification. Technical report, Sun Microsystems, September 2006.

[4] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 316–327, 2005.

[5] C. S. Ananian and M. Rinard. Efficient object-based software transactions. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct 2005. In conjunction with OOPSLA'05.

[6] C. Artho. Finding faults in multi-threaded programs. Master's thesis, ETH Zürich, 2001.

[7] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, 1998.

[8] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[9] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[10] C. Blundell, E. Lewis, and M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. *Workshop on Duplicating, Deconstructing, and Debunking (WDDD), June*, 2005.

[11] C. Boyapati. *Safejava: a unified type system for safe programming*. PhD thesis, Massachusetts Institute of Technology, 2004.

[12] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: A Language for Programming High-Performance Servers. In *Proceedings of USENIX Annual Technical Conference*, pages 129–142, Boston, MA, May 2006.

[13] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM Press.

[14] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 2007. To appear. Available online at http://www.highproductivity.org/HPPLM/final-chamberlain.pdf.

[15] B. Chan and T. S. Abdelrahman. Run-time support for the automatic parallelization of java programs. *J. Supercomput.*, 28(1):91–117, 2004.

[16] P. Charles, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, 2005.

[17] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, 1999.

[18] D. Cunningham. Path inference for atomic sections; first year report. Available online at http://www.doc.ic.ac.uk/~dc04/1st_year_report.pdf, 2006.

[19] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe types for race safety. 2006.

[20] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. *Proc. International Symposium on Distributed Computing*, 2006.

[21] R. Ennals. Software transactional memory should not be obstruction-free. 2006.

[22] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.

[23] Everything2.com. Lock convoying, February 2006. Available online at http://everything2.com/index.pl?node_id=1786627 (accessed 25-December-2006).

[24] P. Felber and M. Reiter. Advanced concurrency control in java. *Concurrency and Computation: Practice and Experience*, 14(4):261–285, 2002.

[25] C. Flanagan and S. Freund. Automatic Synchronization Correction. *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[26] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.

[27] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM Press.

[28] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[29] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[30] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, New York, NY, USA, 2004. ACM Press.

[31] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003.

[32] K. Fraser and T. Harris. Concurrent Programming without Locks. *Submitted for publication*, 2004.

[33] S. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[34] B. Goetz. Synchronization optimizations in mustang. *Java theory and practice (IBM developerWorks)*, October 2005. Available online at http://www-128.ibm.com/developerworks/java/library/j-jtp10185/ (accessed 30-12-2006).

[35] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[36] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. *Granularity of locks and degrees of consistency in a shared data base*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[37] A. Greenhouse and J. Boyland. An object-oriented effects system. *ECOOP*, pages 205–229, 1999.

[38] D. Grossman. Type-safe multithreading in cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.

[39] D. Grossman. Software transactions are to concurrency as garbage collection is to memory management. Technical report, University of Washington, April 2006.

[40] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL'05)*, 2005.

[41] T. Harris. Design choices for language-based transactions. *University of Cambridge Computer Laboratory Tech. Rep., Aug*, 2003.

[42] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[43] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.

[44] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

[45] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, 2006.

[46] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, pages 175–190, 2004.

[47] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[48] M. Herlihy. Sxm1.1: Software transactional memory package for c#. Available online at http://www.cs.brown.edu/~mph/, June 2005.

[49] M. Herlihy, J. Eliot, and B. Moss. Transactional Memory: Architectural Support For Lock-free Data Structures. *Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on*, pages 289–300, 1993.

[50] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 522–529, 2003.

[51] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 253–262, 2006. Library can be downloaded from http://www.sun.com/download/products.xml?id=453fb28e.

[52] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

[53] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[54] B. Hindman and D. Grossman. Strong Atomicity for Java Without Virtual-Machine Support. 2006.

[55] B. Hindman and D. Grossman. Atomicity via source-to-source translation. *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91, 2006.

[56] M. Hoskins. A java framework for building data-intensive applications. White paper, Pervasive Software, 2006.

[57] D. Hovemeyer and W. Pugh. Finding concurrency bugs in java, 2004.

[58] M. Jones. What really happened on mars rover pathfinder. *ACM Forum on Risks to the Public in Computers and Related Systems*, 19(49), December 1997. Available online at http://catless.ncl.ac.uk/Risks/19.49.html#subj1.

[59] D. Kalinsky and M. Barr. Priority inversion. *Embedded Systems Programming*, pages 55–56, April 2002. Available online at http://netrino.com/Publications/Glossary/PriorityInversion.html.

[60] S. Kumar, M. Chu, C. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, 2006.

[61] D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309, 2005.

[62] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[63] S. Lee and R. Liou. A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):144–156, 1996.

[64] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[65] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[66] D. Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGOPS Operating Systems Review*, 11(2):128–137, 1977.

[67] V. Marathe, W. Scherer, and M. Scott. Design tradeoffs in modern software transactional memory systems. *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–7, 2004.

[68] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *CACM*, 3:184–195, 1960.

[69] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006.

[70] M. Moir. Transparent Support for Wait-Free Transactions. *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

[71] K. Moore, M. Hill, and D. Wood. Thread-level transactional memory. *TR1524, Comp. Science Dept. UW Madison, March*, 31, 2005.

[72] J. Moss. Open Nested Transactions: Semantics and Support. *Workshop on Memory Performance Issues*, 2006.

[73] J. Moss and A. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[74] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[75] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[76] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[77] K. Poulsen. Tracking the blackout bug. *Security Focus*, April 2004. Available online at http://www.securityfocus.com/news/8412.

[78] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, December*, pages 01–05, 2001.

[79] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 494–505, 2005.

[80] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill Boston, 2003.

[81] R. M. Ramanathan. Intel multi-core processors: Making the move to quad-core and beyond. White paper, Intel, 2006. Available online at http://www.intel.com/technology/architecture/downloads/quad-core-06.pdf.

[82] M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, 2005.

[83] B. Saha, A. Adl-Tabatabai, R. Hudson, C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, 2006.

[84] W. Scherer III and M. Scott. Advanced contention management for dynamic software transactional memory. *Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 240–248, 2005.

[85] M. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, 1987.

[86] N. Shavit and D. Touitou. Software transactional memory. *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.

[87] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, March 2005. Available online at http://www.gotw.ca/publications/concurrency-ddj.htm.

[88] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[89] C. Szydlowski. Multithreaded technology and multicore processors. *Dr. Dobb's Journal*, May 2005. Available online at http://www.ddj.com/dept/architect/184406074.

[90] M. Voss, editor. *OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools, WOMPAT 2003, Toronto, Canada, June 26-27, 2003, Proceedings*, volume 2716 of *Lecture Notes in Computer Science*. Springer, 2003.

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

[92] A. Welc, A. Hosking, and S. Jagannathan. Transparently Reconciling Transactions with Locking for Java Synchronization. *European Conference on Object-Oriented Programming*, 2006.

[93] Wikipedia. Lock convoy, December 2006. Available online at http://en.wikipedia.org/wiki/Lock_convoy (accessed 25-December-2006).

[94] Wikipedia. Manual memory management, December 2006. Available online at http://en.wikipedia.org/wiki/Manual_memory_management (accessed 24-December-2006).

[95] Wikipedia. Red-black tree, January 2007. Available online at http://en.wikipedia.org/wiki/Red-Black_tree (accessed 14-January-2007).

Appendix A

`Singlestep` toy language grammar

A.1 Declarations

program → (classdecl)* maindecl **EOF**
classdecl → **CLASS IDENT LBRACE** (methoddecl | instvardecl **SEMI**)* **RBRACE**
maindecl → **MAIN** statlist
methoddecl → constructordecl | type **IDENT LPAREN** (paramslist)? **RPAREN** statlist
constructordecl → **IDENT LPAREN** (paramslist)? **RPAREN** statlist
paramslist → param (**COMMA** param)*
param → type **IDENT**
vardecllist → vardecl (**COMMA** vardecl)*
instvardecl → type **IDENT**
vardecl → type **IDENT** (**ASSIGN** expr)?
type → **INT** | **BOOL** | **STR** | **IDENT** | **VOID**

A.2 Statements

statlist → **LBRACE** (stat)? **RBRACE**
stat → expr **SEMI** | vardecl **SEMI** | statlist | ifstat | whilestat | atomicstat | returnstat | spawnstat | printstat | lockstat | joinstat | skipstat
ifstat → **IF LPAREN** expr **RPAREN** stat **ELSE** stat
whilestat → **WHILE LPAREN** expr **RPAREN** stat
atomicstat → **ATOMIC** stat
returnstat → **RETURN** (expr)? **SEMI**
spawnstat → **SPAWN** (vardecllist)? statlist
lockstat → (**LOCKR** | **LOCKRW** | **UNLOCKR** | **UNLOCKRW** | **UNLOCK**) **LPAREN** path **RPAREN SEMI**
joinstat → **JOIN SEMI**
skipstat → **SKIP SEMI**
printstat → **PRINT** (expr | **STRING**) **SEMI**

A.3 Expressions

exprlist → expr (**COMMA** expr)*
expr → assignexpr

assignexpr → path **ASSIGN** assignexpr | orexpr
orexpr → andexpr (**OR** andexpr)*
andexpr → equalexpr (**AND** equalexpr)*
equalexpr → relexpr ((**EQ** | **NEQ**) relexpr)*
relexpr → addexpr ((**LT** | **LTE** | **GT** | **GTE**) addexpr)?
addexpr → mulexpr ((**PLUS** | **MINUS**) mulexpr)*
mulexpr → unaryexpr ((**TIMES** | **DIV** | **MOD**) unaryexpr)*
unaryexpr → (**INC** | **DEC** | **PLUS** | **MINUS**) unaryexp
| **NOT** postfixexpr | postfixexpr
postfixexpr → primaryexpr (**INC** | **DEC**)?
primaryexpr → pathormethod | newexpr | **TRUE** | **FALSE** | **NULL** | **THIS** | **STRING**
| **INTEGER** | **LPAREN** expr **RPAREN**
pathormethod → path (**LPAREN** (exprlist)? **RPAREN**)?
path → **IDENT** (**DOT IDENT**)*
newexpr → **NEW IDENT LPAREN** (exprlist)? **RPAREN**

A.4 Tokens

CLASS → "class"
MAIN → "main"
IF → "if"
ELSE → "else"
WHILE → "while"
ATOMIC → "atomic"
RETURN → "return"
SPAWN → "spawn"
LOCKR → "lockr"
LOCKRW → "lockrw"
UNLOCKR → "unlockr"
UNLOCKRW → "unlockrw"
UNLOCK → "unlock"
JOIN → "join"
SKIP → "skip"
PRINT → "print"
NEW → "new"

LBRACE → '{'
RBRACE → '}'
LPAREN → '('
RPAREN → ')'
IDENT → ('a' - 'z') ('a' - 'z' 'A' - 'Z' '0' - '9')*
INTEGER → '0' - '9'
STRING → \"(**IDENT**)*\"
COMMA → ','
DOT → '.'
ASSIGN → '='
SEMI → ';'

OR → "||"
AND → "&&"
EQ → "=="
NEQ → "!="
LT → '<'
LTE → "<="
GT → '>'
GTE → ">="
PLUS → '+'
MINUS → '-'
TIMES → '*'
DIV → '/'
MOD → '%'
INC → "++"
DEC → "- -"
NOT → '!'
TRUE → "true"
FALSE → "false"
NULL → "null"
THIS → "this"
INT → "int"
BOOL → "bool"
STR → "string"
VOID → "void"