Improving the performance of Atomic Sections

Transfer Report

Khilan Gudka

First Supervisor: Professor Susan Eisenbach
Second Supervisor: Professor Sophia Drossopoulou

June 5, 2009

Abstract

Atomicity is an important property for concurrent software, as it provides a stronger guarantee against errors caused by unanticipated thread interactions than race-freedom does. However, concurrency control in general is tricky to get right because current techniques are too low-level and error-prone. With the introduction of multicore processors, the problems are compounded. Consequently, a new software abstraction is gaining popularity to take care of concurrency control and the enforcing of atomicity properties, called atomic sections.

One possible implementation of their semantics is to acquire a global lock upon entry to each atomic section, ensuring that they execute in mutual exclusion. However, this cripples concurrency, as non-interfering atomic sections cannot run in parallel. Therefore, from a language designer's point of view, the challenge is to implement atomic sections without compromising performance.

This thesis explores the technique of lock inference, which infers a set of locks that attempt to balance the requirements of maximal concurrency, minimal locking overhead and freedom from deadlock. In particular, we look to improve upon the current state-of-the-art as well as consider newer territory such as parallelism within an atomic section to exploit multicore processors and a hybrid implementation with transactional memory.

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
  - 1.1.1 Subtleties of concurrent programming . . . 5
  - 1.1.2 Preventing race-conditions . . . 6
  - 1.1.3 Race-freedom as a non-interference property . . . 7
  - 1.1.4 Enter the world of atomicity . . . 8
  - 1.1.5 The ~~*complexities*~~ joys of locks . . . 8
  - 1.1.6 Incompleteness . . . 9
  - 1.1.7 A better abstraction for concurrency . . . 9
  - 1.1.8 Implementing atomic sections . . . 10
- 1.2 Contributions . . . 11

2 Background **13**
- 2.1 Semantics of atomic sections . . . 13
- 2.2 Related areas . . . 14
  - 2.2.1 Transactional memory . . . 14
  - 2.2.2 Program analysis . . . 15
- 2.3 Literature review . . . 20
  - 2.3.1 Basics of lock inference . . . 20
  - 2.3.2 Inferring shared accesses . . . 21
  - 2.3.3 Inferring locks . . . 25
  - 2.3.4 Acquiring/releasing locks . . . 26
  - 2.3.5 Additional features . . . 27
- 2.4 Soot . . . 28

3 Progress **29**
- 3.1 Scaling the approach to Java . . . 29
- 3.2 Summaries . . . 30
  - 3.2.1 Transformers . . . 30
- 3.3 Next steps . . . 30

4 Research Plan **31**
- 4.1 Time-scale . . . 31
- 4.2 Deliverables . . . 32

Bibliography **37**

Chapter 1

Introduction

1.1 Motivation

Processor manufacturers can no longer continue to increase clock speeds at the same rate they have done previously, due to the demands it places on power [44]. Hence, they are now using increases in transistor density, as predicted by Moore's law, to put multiple processing cores on a chip. Furthermore, this trend is likely to continue for the foreseeable future. For example, Intel predicts that by 2011, processors with 80-cores will be ready for commercial production [35].

In order to harness such parallel computing power as well as continue to get free increases in software performance from increases in hardware performance, software programs need to be concurrent [58, 59]. That is, structured as a set of logical activities that execute simultaneously. For example, a web server consists of a number of workers enabling it to accept and process multiple client requests at the same time.

At present, the vast majority of programs are sequential [58], performing only one logical activity at any one time. One reason for this might be the lack of true parallelism, however, a fundamentally more serious problem is that **concurrent programming with current techniques is inherently difficult and error-prone** [47]. We inevitably need better abstractions.

1.1.1 Subtleties of concurrent programming

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, to keep them lightweight, they share their owning process's resources, including its address space. 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 is controlled, it can lead to interference, more commonly referred to as a *race condition* [47]. This occurs when two or more threads access the same memory location and at least one of the accesses is a write.

Figure 1.1 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 1.1(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.

Figure 1.1: *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)

As a result, a race condition occurs and an 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 [37, 32, 49].

1.1.2 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 datum 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 performing conflicting operations at the same time and thus interfering with

Figure 1.2: *Race free version of the example given in Figure 1.1.*

```
class Counter {
  int counter = 0;

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

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

each other.

Figure 1.2 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 receiver object before proceeding. If the lock is currently unavailable, the requesting thread is blocked and placed into 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.

1.1.3 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. To illustrate this, we extend our `Counter` class to include a method `reset`, which resets the value of the counter to that provided as an argument to it. Moreover, it is declared `synchronized` to prevent races.

Figure 1.3(a) shows the updated `Counter` class as well as an example scenario involving two counters (`c1` and `c2`) and two threads (`T1` and `T2`). Thread `T1` wishes to reset both counters with the value 1, while `T2` proceeds to reset them with value 2. It is worth noting here that the intention is that both counters are reset together, regardless of the order in which the threads are run. That is, whether T1's double reset persists or T2's is a matter of timing. However, we want the resets to be performed in a pair. Figure 1.3(b) gives an example interleaving of their calls to `reset`. `T1` begins by resetting counter `c1` to 1 but is preempted before it can update `c2`. Thread `T2` is then run to completion. At this point, both counters have the value 2, which is a valid outcome of our execution. However, `T1` resumes and resets `c2` to 1. The final result is that `c1.counter` is 2 and `c2.counter` is 1. This does not represent `T1`'s intention nor `T2`'s.

Such incorrect behaviour occurs because thread `T2` is able to modify counter `c2` while `T1` is performing its double reset. This is possible because although `T1`'s invocations of `c1.reset(1)` and `c2.reset(1)` individually ensure mutually exclusive access to `c1` and `c2` respectively, their composition does not. As a result, `T2`'s operations can be interleaved between them leading to the higher-level interference. Note that there are no races, as `reset` is declared `synchronized`.

Figure 1.3: *An example illustrating that asserting race-freedom is not enough to ensure freedom from all errors caused by thread interactions.*

```
class Counter {
  //...as above...
  synchronized void reset(int val) {
    counter = val;
  }
}
Counter c1 = new Counter();
Counter c2 = new Counter();

Thread T1:
  c1.reset(1);
  c2.reset(1);

Thread T2:
  c1.reset(2);
  c2.reset(2);
```

(a)

	T1	*T2*
1	c1.reset(1)	
2		c1.reset(2)
3		c2.reset(2)
4	c2.reset(1)	

c1.counter	*c2.counter*
2	1

(b)

1.1.4 Enter the world of atomicity

To assert that such interferences do not occur, we need a stronger property that ensures that threads cannot interleave conflicting operations while a block of code is executing, that is the *atomicity of code blocks*. A code block is said to be *atomic* if the result of any concurrent execution involving it is equivalent to the sequential case. This means that in our example of Figure 1.3, the result of `T1` and `T2` executing concurrently would be the same as if `T1` and `T2` executed one after the other, leaving both counters either with value 1 or 2. Atomicity is a very powerful concept, as it enables us to reason about a program's behaviour at a simpler level. It abstracts away the interleavings of different threads (even though in reality, interleaving will still occur) enabling us to think about a program sequentially.

A number of techniques exist to verify atomicity of code blocks such as: type checking [15, 14], type inference [13], model checking [27], theorem proving [19] and run-time analysis [12, 61]. However, enforcing atomicity is still left to the programmer, usually using locks.

1.1.5 The ~~*complexities*~~ joys of locks

In order to make the double resets of `T1` and `T2` in Figure 1.3 atomic, each thread would need to first acquire locks on both counters before proceeding. This would prevent the other thread from interleaving conflicting calls to `reset`. Figure 1.4(a) gives one possible implementation.

However, Figure 1.4(b) shows yet another problem that arises from this race-free and atomic version. Thread `T1` acquires the lock on `c1` but is preempted before it tries to lock `c2`. `T2` is then allowed to run, however, it decides to acquire the locks in the opposite order. It locks `c2` and attempts to lock `c1` but cannot as `T1` has already done so. Subsequently, `T2` is blocked waiting for `T1` to release the lock on `c1`. `T1` resumes and attempts to lock `c2`, however cannot

Figure 1.4: *A version of Figure 1.3, whereby the compound resets of each thread occur in mutual exclusion, enforced using locks. However, this example illustrates the additional problem of deadlock.*

```
Thread T1:
synchronized(c1) {
  synchronized(c2) {
    c1.reset(1);
    c2.reset(1);
  }
}
```

```
Thread T2:
synchronized(c2) {
  synchronized(c1) {
    c1.reset(2);
    c2.reset(2);
  }
}
```

(a)

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

(b)

because `T2` holds the lock. As a result, it is blocked waiting for `T2` to release `c2`. Hence, we now have a situation whereby both threads are stuck: `T1` is waiting for `T2` to release the lock on `c2` and `T2` is waiting for `T1` to release the lock on `c1`. Neither thread can continue and the system has come to a standstill. This state is known as *deadlock*.

Other problems that can occur due to locks include priority inversion [33], livelock [16] and convoying [10, 63]. Furthermore, forgetting to acquire a lock leads round-circle back to the problem of races again. The worst part is that these problems are hard to detect at compile-time and their impact at run-time can be disastrous [32, 37, 49].

1.1.6 Incompleteness

In addition to the problems that arise when trying to ensure atomicity, it may not actually always be possible to do so. 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 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*. One 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.

In summary, although atomicity allows us to more confidently assert the absence of errors due to thread interactions, 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.

1.1.7 A better abstraction for concurrency

This has led researchers to a language-level abstraction for enforcing the atomicity of a group of statements. *Atomic sections* [40] are blocks of code that appear to other threads to execute in a single step, with the details of how this is achieved being taken care of by the compiler and/or run-time. This is a significant improvement over current abstractions, as atomic sections aim to completely relieve the programmer from worrying about concurrency control and consequently eliminate the associated complexities. They enable programmers to think in terms of single-threaded semantics, also removing the need to make classes/libraries thread safe. Furthermore,

Figure 1.5: *An implementation of the* `Counter` *class using atomic sections.*

```
class Counter {
  int counter = 0;

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

  atomic void reset(int val) {
    counter = val;
  }
}
```

(a)

```
Thread T1:
  atomic {
    c1.reset(1);
    c2.reset(1);
  }

Thread T2:
  atomic {
    c1.reset(2);
    c2.reset(2);
  }
```

(b)

error handling is considerably simplified because code within an atomic section is guaranteed to execute without interference from other threads making error recovery 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 and in what order, as protecting them and avoiding deadlock is taken care of by the 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, programmer intent is mostly atomicity when using locks [15], hence atomic sections enable programmers to more accurately specify their intentions. Figure 1.5 shows an implementation of our double-counter-reset example using atomic sections (denoted using the `atomic` keyword). Moreover, note that there is no longer the potential for deadlock.

1.1.8 Implementing atomic sections

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

- **Interrupts:** Proposed in Lomet's seminal paper [40], whereby interrupts are disabled while a thread executes inside an atomic section.
- **Co-operative scheduling:** Involves intelligently scheduling threads such that their interleavings ensure atomicity [56].
- **Object proxying:** A very limited technique whereby proxy objects are used to perform lock acquisitions before object invocations at runtime [11].
- **Transactional memory:** Atomic sections are executed as database-style transactions. In particular, memory updates are buffered until the end of the atomic section and subsequently committed in 'one step' if conflicting updates have not been performed by other threads. Otherwise, the changes are rolled back (i.e. the buffer is discarded) and the atomic section is re-executed [36].

- **Lock inference:** An approach that statically infers which locks need to be acquired to ensure atomicity and transparently inserts acquire and release statements in such a way that deadlock is avoided [29, 41, 5, 21, 6, 8, 65].

- **Hybrids:** Approaches that combine several of the above techniques. For example, using locks when there is no contention or when an atomic section contains an irreversible operation, and transactions otherwise [62].

While nobody yet knows what is the best way of implementing atomic sections, transactional memory seems to be the most popular approach. However, it has a number of drawbacks, including: poor support for irreversible operations such as I/O and system calls, high run-time overheads in both contended and uncontended cases and a large amount of wasted computation.

Lock inference is a promising alternative, as it has a number of advantages over transactional memory: firstly, it doesn't limit expressiveness, secondly, it provides excellent performance in the common case of where there is no contention and thirdly, it has little run-time overhead. Initially, it may seem that we are re-inviting the problems associated with locks, however, a combination of static analyses and run-time support are typically used to overcome them.

There are a number of challenges that this approach faces:

- Maximising concurrency

- Ensuring freedom from deadlock

- Minimising the number of inferred locks

However, these three goals can be in conflict. For example, to maximise concurrency, the locks should be fine-grained and the number of locks should scale with the number of objects. However, this increases the run-time overhead caused by acquire and release operations. Furthermore, there is now a greater chance of deadlock. On the other hand, we could use a single global lock to protect all atomic sections, acquired on the entry and released on the exit of each. This gives us a minimal number of locks and eliminates deadlock, however, it prevents any concurrency whatsoever. Hence, we need to strike a balance. Moreover, this balance will probably differ depending on the application. This thesis will look at extending work done in this area.

1.2 Contributions

During the PhD, I hope to make the following contributions:

- Improve existing lock inference techniques to allow concurrency in scenarios where they fail.

- Formally investigate whether the semantics given by a lock inference implementation matches the higher-level semantics expected by the programmer using atomic sections. In particular, we ask the question of whether they adhere to the memory model when accesses to an object occur both within and outside an atomic section.

- Look into additional ways of improving the performance and expressiveness supported by lock inference:

  - Supporting parallelism, such as the fork-join model, within the atomic section. This area has not yet been looked at by existing lock inference work.

– A hybrid lock inference/transactional system, which may use dynamic feedback or heuristics to determine where locks or transactions would be more beneficial. Current approaches typically execute existing monitors, such as those declared using `synchronized`, transactionally or using the associated lock. However, nothing has been done in conjunction with inferring locks.

Chapter 2

Background

Atomic sections were first proposed by Lomet in his 1977 paper [40]. However, they have only recently come into the forefront of programming language research. The last five years in particular have seen a huge upsurge in interest, with the majority of contributions being made in the sub-area of transactional memory. Lock inference has also attracted contributions and is seen as an important alternative and perhaps ultimately a complementary approach. In fact, the most effective implementation of atomic sections will probably involve a marriage of the two techniques.

In this chapter, we will review the relevant literature giving focus to lock inference rather than transactional memory, as the former is the basis of the PhD. However, we begin by looking more closely at the semantics of atomic sections.

2.1 Semantics of atomic sections

Conceptually, atomic sections execute as if in 'one 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 [36, 4]:

- **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 only guaranteed with respect to other atomic sections.

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 it may be too high thus resulting in a trade off between performance and ease of programming. Although, a number of optimisations have been proposed for transactional memory to reduce this overhead [2, 30, 55], with [2] reporting performance within 25% of an implementation only guaranteeing weak atomicity.

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 strengthen it by using a static analysis to detect conflicting shared accesses occurring outside atomic sections and subsequently wrap them inside `atomic{}`. Recent work [1] has looked at a dynamic approach which verifies, at run-time, that atomic accesses of a datum never coincide

with non-atomic accesses of it. That is, during its life-time, it can be accessed both inside atomic sections (termed a *protected* access) and outside (termed an *unprotected* access) but never both simultaneously. If while in protected mode, an unprotected access never occurs; and while in unprotected mode, a protected access never occurs, then the program will run with strong atomicity semantics. They call this *dynamic separation*.

Lock inference techniques unanimously assume that no shared accesses occur outside atomic sections, consequently avoiding this debate. Although, a static analysis like that described above could be used.

2.2 Related areas

In this section, we briefly review related areas.

2.2.1 Transactional memory

Transactional memory is a form of optimistic concurrency control whereby the key idea is to buffer memory updates during execution and only perform them on memory at the end if the locations that were accessed have not been modified by another thread. If they have, the would-be updates are discarded (termed *rollback*) and the transaction is re-executed [25].

To prevent interference while a transaction is updating memory (termed *commit*), it first acquires ownership of the locations to be updated. If a location is owned by another transaction, it will rollback and be re-executed. Ownership is released when the updates have been completed. In this way, atomicity is achieved without blocking threads [24].

Transactional memory provides a number of advantages over traditional blocking primitives such as locks:

- **No deadlock, priority inversion or convoying:** as there are no locks! Although, in theory a different form of priority inversion could 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. Transactional memory provides the finest possible granularity (at the memory-word level), resulting in optimal parallelism. However, this comes at the cost of buffering overheads.
- **Automatic error handling:** Memory updates are automatically undone upon rollback, reducing the need for error handling code [22].
- **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.

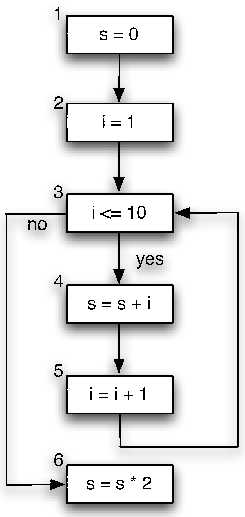
However, these advantages rely on being able to rollback. This proves to be a huge limitation for atomic sections as it leads to the following drawbacks:

- **Irreversible operations:** Transactions cannot contain irreversible operations such as I/O because they cannot be rolled-back. Buffering is one proposed solution [23, 30], but requires rewriting I/O libraries and is not applicable in all situations. For example, consider a handshake with a remote server. Some implementations forbid irreversible actions using the type system [25], while others throw exceptions [50]. However, these are not practical in general.

Figure 2.1: *(a) is a program that calculates double the sum of 1 to 10. (b) is its control flow graph.*

```
s = 0;
i = 1;
while(i <= 10) {
    s = s + i;
    i = i + 1; }
s = s * 2;
```

(a)



(b)

- **Performance overhead:** Significant overhead is incurred due to buffering, validating that accessed locations have not been modified and committing [24].
- **Wasted computation:** A transaction that is later rolled-back is wasted computation. In one benchmark [31], tens of rollbacks occurred per second.

Performance has been the main focus of transactional memory research over the last few years. A lot of progress has been made, such as removing the requirement that transactions be non-blocking [7, 9, 26, 31, 53], coarsening the granularity from memory-word to object [42, 18, 17, 28, 3, 31, 26] and making application-specific the decision made when a transaction detects that its accessed locations have been modified [54].

However, current state-of-the-art implementations still require rollback for avoiding deadlock and starvation [31]. Furthermore, the problem of I/O remains.

2.2.2 Program analysis

Program analysis allows us to approximate run-time behaviours of programs at compile-time. It is used in lock inference to determine which objects are being accessed and what their corresponding locks are. This section provides a very brief overview of relevant concepts. For a detailed account, please refer to [46].

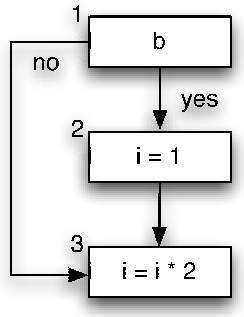
2.2.2.1 Data flow analysis

The approach to program analysis that is of relevance to this project is *data flow analysis*. In this technique, it is customary to think of a program as a graph: the nodes are simple statements or expressions and the edges describe how control might pass from one simple statement to another. This is called a *control flow graph*. Figure 2.1(b) shows an example graph for a program that calculates double the sum of 1 to 10. Nodes are labelled uniquely. Notice the two edges coming out of the while condition corresponding to where flow goes to when it is true

Figure 2.2: *Simple program to demonstrate the difference between may and must analyses.*

```
int i;
if(b)
    i = 1;
i = i * 2;
```

(a)



(b)

or false. At compile-time, we typically cannot determine exactly which edges will be followed, therefore we must consider all of them. 'If' statements are similar.

In a nutshell, data flow analysis works by pushing sets of 'things' through the graph until they stabilise. When a node receives data from immediately preceding nodes, called *entry information*, it applies the effect of its statement and passes the resulting set, called *exit information*, to its immediate successors. If a node has multiple predecessors, like 3 in Figure 2.1(b), the incoming data are first combined using set union or intersection depending on the type of analysis.

There are broadly four types of data flow analyses depending on whether (a) we want information that is valid along all paths from the start of the program to a node or only some of them and (b) we want to know something about code before nodes or after them. For (a), consider the example given in Figure 2.2 and suppose we want to know whether variable `i` has been initialised before reaching 3. The start node of the program is 1. There are two paths from 1 to 3: $1 \to 2 \to 3$ and $1 \to 3$. `i` is initialised along the first but not the second. Therefore, we deduce `i` may not be initialised. This is called a *must analysis* because we only assert `i` is initialised if all paths from 1 to 3 initialise `i`. The key point here is that we consider all paths. In this type of analysis, data from immediate predecessors are combined using set intersection. If instead we wish to determine what value `i` might have, we union the result of each path. This is called a *may analysis*. In this, data from immediate predecessors are combined using set union.

Sometimes we will want to calculate information about paths reaching a node and other times about paths leaving a node. For example, determining if `i` is initialised at 3 in Figure 2.2 requires looking at paths reaching it. On the other hand, determining what objects are accessed in an atomic section, requires looking at paths leaving the start of the section. In the former, data is passed from the start of the program downwards. This is called a *forwards analysis*. In the latter, data is passed from the end of the program upwards. This is called a *backwards analysis*. Note that the predecessor of a node in a forwards analysis will be the successor of a node in a backwards analysis. Do not confuse this with control flow, we are talking about *data flow*.

This leads to the following four types of data flow analyses:

- Forwards, must
- Forwards, may

Figure 2.3: *Iterative algorithm for computing entry and exit sets.*

```
while(entry and exit sets change) {
    for each node n {
      // calculate new entry set
      entry'(n) = { };
      for each predecessor node p of n
          entry'(n) = entry'(n) + exit(p);

      // calculate new exit set
      exit'(n) = f_n(entry(n)); } }
```

- Backwards, must
- Backwards, may

Entry and exit information are commonly referred to as the entry and exit sets of the node. In a forwards analysis, the entry sets give us the final information we want, while in a backwards analysis it is the exit sets. Note that while the notion of predecessor and successor get swapped around, entry and exit sets do not. That is, in a backwards analysis the exit set is calculated by combining the results of its predecessors and the entry set is calculated by pushing this data 'through' the node. This implies that the notion of 'entry' and 'exit' remain consistent with the control flow graph.

To calculate the final entry and exit sets, an iterative algorithm is used. Figure 2.3 gives it in pseudo-code.

Here we use '′' to distinguish between the current and previous iterations. The function f_n applies the effect of n's statement to its previous entry set. It is known as a *transfer function*. This function will typically kill some incoming data and add any additional information created by this node. These are called its *kill* and *gen* sets respectively. One can express f_n in terms of these sets as follows:

$$f_n(d) = (d \setminus kill_n(d)) \cup gen_n(d)$$

The algorithm terminates when every entry and exit set does not change between iterations. This is referred to as having reached a *fixed point*.

2.2.2.2 Intraprocedural versus interprocedural

So far we have only looked at data flow analysis in a single method. This is known as *intraprocedural data flow analysis*. Lock inference also needs to determine object accesses in methods called from atomic sections because these need to be protected too. When we consider data flow across methods, this is called *interprocedural data flow analysis*.

The key idea is that data to a node n that performs a method call (called a *caller node*) flows to the start of the corresponding method m and exit information from m's last node flows back to n. Calculating the entry set is the same as in the intraprocedural case, but the exit set is now calculated from both the entry set and the information flowing back from m. Figure 2.4 gives a graphical description. Here, d is the entry information for n and d' is the data flowing back from m.

Figure 2.4: *Interprocedural analysis.*

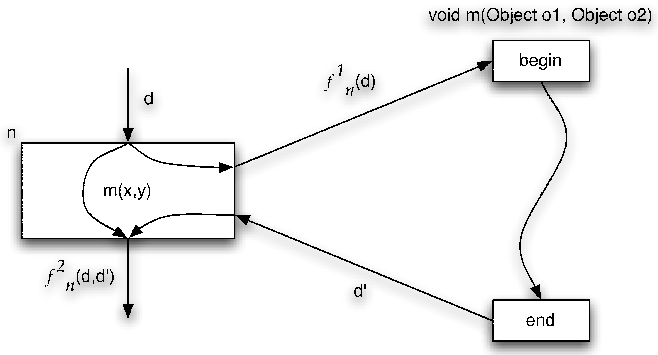
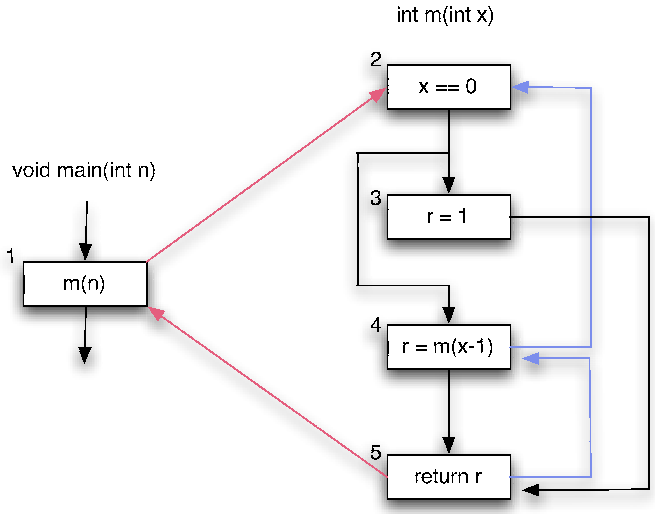


Figure 2.5: *Problem of valid paths.*

```
int m(int x) {
    int r;
    if(x == 0)
        r = 1;
    else
        r = m(x-1);
    return r;
}

void main(int n) {
    m(n);
}
```

(a)



(b)

Interprocedural analysis introduces two new functions $f_n^1(d)$ and $f_n^2(d, d')$. f_n^1 modifies the incoming data as required for passing to the method. This might include removing information about local variables and renaming arguments to the corresponding formal parameter names. f_n^2 modifies the data flowing back from the method as appropriate for returning from it and combines it with the entry information for n. The former might include renaming formal parameters. The entry information may also be modified based on the returning data.

2.2.2.3 Valid paths through the program

Armed with these two functions, we could carry out the interprocedural analysis like in the intraprocedural case. However, this turns out to be rather naive because it allows data to flow along paths that do not correspond to a run of the program. Consider the example program in Figure 2.5. At run-time, there will typically be a stack of method calls that are waiting to be returned to. Execution always returns to the most recent one first, i.e. the one at the top of stack. However, notice that in Figure 2.5 there is nothing stopping the analysis from considering

the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 1$ corresponding to calling m twice but returning only once. This is a problem because it can lead to incorrect solutions.

We can restrict consideration to valid paths by associating call stacks with data. This will typically be a string of labels corresponding to method call nodes with the most recent on the right. This string is called a *calling context* [57]. Now, data is a set of functions mapping contexts to the data flow information for that context. Note that we may have several contexts at a time because there may be several ways of reaching a method from the start of the program. The two key things that make using contexts work are:

- Before passing data to a method, append the caller node's label to all contexts. This indicates that it is now the most recent method call.
- For all contexts passed back by the method, only keep those whose right most label is this node. This ensures that we don't pass data back along the wrong paths. To indicate we have returned from the call, remove this right most label.

An analysis that uses contexts is called *context-sensitive*.

2.2.2.4 Summaries

One of the widely known problems with using call strings [34], is that for programs with deep call chains, the number of call strings can be tremendously high. As a result, context-sensitive analyses tend to be very expensive both in terms of time and memory usage especially when analysing large programs. A more widely used alternative, which is still context-sensitive, is the *method summary* approach [57] which involves calculating for each method, a function that describes how the method as a whole translates data flow information. Data flow facts don't have to be flowed through a target method `m` but instead are translated in one step using `m`'s summary function.

A summary function is computed by first defining, for each individual statement, functions describing how they each translate data flow information and then composing them into one large function for the entire method. Essentially, the data flow information during summary computation are these functions. Therefore, the challenge is to find a representation that affords fast composition and meet operations.

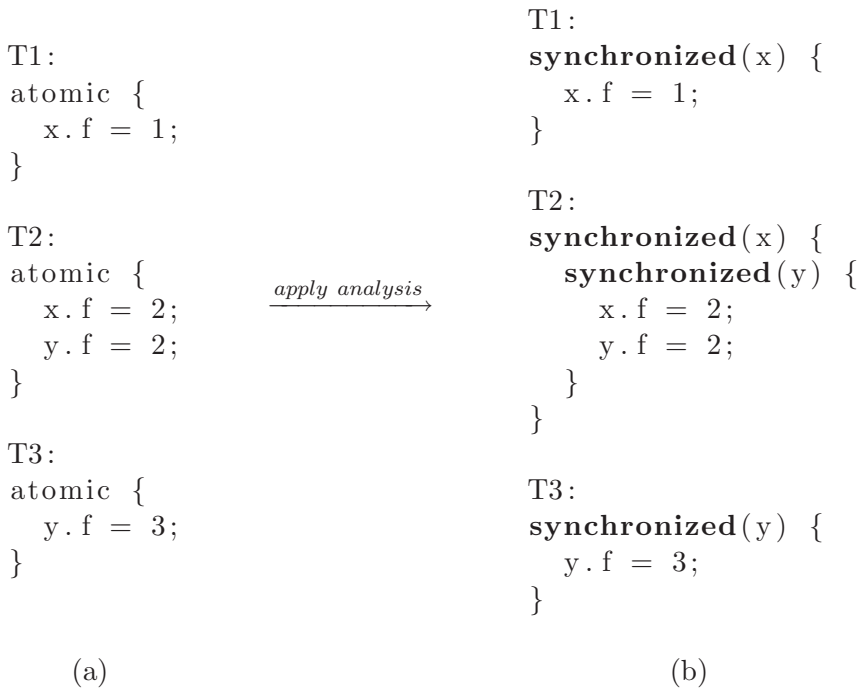
2.2.2.5 IDE analyses

An important category of data flow analyes are called Interprocedural Distributive Environment (IDE) analyses. This is a very general class containing analyses such as copy-constant propagation and linear-constant propagation, object naming analysis, 0-CFA type analysis, and all IFDS (interprocedural, finite, distributive, subset) problems such as reaching definitions, available expressions, live variables, truly-live variables, possibly uninitialised variables, flow-sensitive side-effects, some forms of may-alias and must-alias analysis, and interprocedural slicing [51].

In an IDE problem, data flow facts are environments. That is, they are mappings of the form $D \rightarrow L$ where D is a finite set of symbols D and L is a finite height semi-lattice. For example, in the case of constant-propagation, D would be the set of variables in the program and $L = \{\top, \bot\} \cup \mathbb{Z}$.

The advantage of formulating an analysis in the IDE framework, is that efficient representations have been developed that allow fast composition and meet [52]. In summary, they represent transfer functions, called *transformers*, as graphs. This allows composition to be computed by simply taking the transitive closure.

Figure 2.6: *An example illustrating the general idea behind lock inference.*



2.3 Literature review

We now conduct the review by first defining a unified framework consisting of the dimensions along which existing lock inference work can differ. We then use this framework to compare the contributions, as shown in Figure 2.1.

2.3.1 Basics of lock inference

The general idea behind lock inference, given a program containing atomic sections, is to statically infer a set of locks for each atomic section to acquire and release, which ensure that the result of concurrently executing multiple atomic sections is equivalent to executing them in some serial order. That is, the locks should ensure *serialisability* [41] of all atomic sections.

To illustrate this, consider the example program in Figure 2.6(a). It consists of two global objects `x` and `y`, as well as three threads `T1`, `T2` and `T3` performing concurrent updates to their `f` fields. To avoid interfering with each other, the threads perform their updates inside atomic sections.

Lock inference begins by performing a compile-time analysis to determine what shared accesses are being performed by each atomic section. It then maps these shared accesses to locks, trying to balance the requirements of maximal concurrency, a minimal number of locks and freedom from deadlock. Finally, these locks are inserted into the program in the form of acquire and release operations. In this example, the analysis infers that `T1` accesses `x`; `T2` accesses `x` and `y`; and `T3` accesses `y`. When mapping these accesses to locks, it will notice that `T1` and `T3` perform disjoint accesses and should consequently be allowed to run in parallel by not being given the same lock. Furthermore, `T2` conflicts with both and therefore should have a (different) lock in common with each of `T1` and `T3`. The solution in this case, as shown

Figure 2.7: *Iterating through a dynamic data structure.*

```
Node n = list.head;
while (n != null) {
    n = n.next;
}
```

in Figure 2.6(b), is to protect each global object with its own lock and acquire the lock(s) corresponding to the object(s) accessed by the particular atomic in question. This allows `T1` and `T3` to execute in parallel but serialises `T1` and `T2` as well as `T2` and `T3`. This example uses Java's `synchronized` construct, which acquires the unique lock protecting the argument object and releases it after exiting the block.

2.3.2 Inferring shared accesses

Lock inference proceeds by first inferring what shared accesses are performed by each atomic section. This allows the analysis to determine potential conflicts, which it can mitigate with a suitable set of locks. However, this is complicated by the fact that the number of datum accessed at run-time may not be completely known at compile-time, such as when traversing dynamic data structures like linked lists. Figure 2.7 shows an example.

Lock inference analyses, as they are performed at compile-time, have to represent such potentially infinite sets of accesses in a finite manner. How this is done depends on how the analysis represents data accesses.

2.3.2.1 Data representation

There are two representations inferred by existing lock inference work. One approach is to infer *abstract objects* [29, 21]. An abstract object is an allocation site of the form `new T`. They are called abstract because many run-time objects may be created by the same allocation site. For example:

```
1 Car[] cars = new Car[N];
2 for(int i=0; i<N; i++) {
3   cars[i] = new Car();
4 }
```

This program fragment creates an array with N elements and initialises each one with a new `Car` instance, giving a total of N+1 run-time objects. Furthermore, there are two abstract objects o_1 and o_3, representing the allocations at lines 1 and 3 respectively. While there is a one-to-one mapping between the run-time and compile-time array object `cars`, we have the unfortunate result that all elements in the array are mapped to the same abstract object o_3. Consequently, accesses of distinct array elements will be considered by the analysis as accesses of the same object, resulting in a conflict being detected that doesn't exist. In general, an inference algorithm using this technique determines which of these abstract objects are pointed to by variables and fields inside the atomic section. This is known as a *points-to* analysis [48].

The second approach is to infer *lvalues* [41, 6, 8, 5]. An lvalue is a syntactic expression that refers to an object on the heap. Examples include `x.f.g` (in Java) and `x->f->g` (in C/C++). At run-time, each lvalue can evaluate to any number of objects. For example:

Table 2.1: *Comparison of lock inference approaches (considered in chronological order)*

	McCloskey [41]	Hicks [29]	Emmi [8]	Zhang [65]	Halpert [21]	Cunningham [6]	Cherem [5]
Language	C	C	C, Java	OpenMP	Java	Java	C/C++, C#
Inferring accesses							
Data representation	Lvals	Allocs	Lvals	?	Allocs	Lvals	Lvals
Aliasing	Yes	Yes	Yes	?	Yes	Yes	Yes
Assignment	No	N/A	No	?	N/A	Rewrite	Rewrite
Unbounded accesses	N/A	N/A	N/A	N/A	N/A	Regex	Limit length
Local/shared	Yes*	Yes	No	No	Yes	No	No
Inferring locks							
Isolate conflicts	No	No	No	Yes	Yes	No	No
Isolate concurrency	No	No	No	No	Yes	No	No
Data to locks	Manual	Auto	Auto	Auto	Auto	Auto	Auto
Lock minimisation	None	Coalesce	ILP	ILP, Heuristics	Heuristics	None	None
Locking granularity	Static/Dynamic	Static	Static/Dynamic	Static	Static/Dynamic	Multigrain	Multigrain
Acquiring/releasing locks							
Locking policy	Strict 2PL	Basic 2PL	Strict 2PL	Basic 2PL	Basic 2PL	Early unlocking	Basic 2PL
Deadlock	Static	Static	Static	Static	Static	Dynamic	Dynamic?
Additional features							
True nesting	No	No	No	No	Yes	No	No
Condition variables	Yes	No	No	Yes	Yes	No	No
Evaluation							
Large examples	Yes	No	Yes	Yes	Yes	No	Yes
Run-time results	Yes	No	No	Yes	Yes	No	Yes

Figure 2.8: *Assignments (a) and aliasing (b) affect which lvalues are inferred.*

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

(a)

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

(b)

```
public void m(A a) {
  a.f = 1;
}
```

In this example, method `m` takes a parameter of type `A` and modifies its `f` field. With abstract objects, we infer all allocations that could be pointed to by `a`, whereas the lvalues approach infers the expression `a`. Note that during the life-time of the program, `a` may point to an unbounded number of objects, however, if the (possibly unique) lock used to protect each such object is somehow reachable from the object; that is, it can be expressed as an extension of the lvalue, such as `a.lock`, then we can lock each of these objects individually. This is much finer-grained than when using abstract objects because there the maximum number of locks is bounded by the number of such objects known at compile-time.

2.3.2.2 Assignment

Lvalues can be assigned to one or more times in an atomic section. As a result, the object being referred to at an access may not be the same as where locks are acquired. Consider the example in Figure 2.8(a). The object being updated in the second line is `me.account`. However, `me.account` is assigned to `you.account` before the update. Hence, with respect to the start of the atomic section, the object being updated is actually `you.account`.

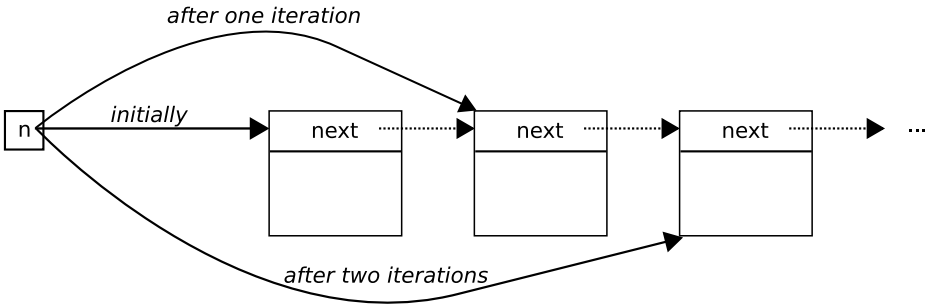
In [6, 5], lvalues are rewritten as they are pushed up the atomic section while [41] forces the lock aquisition to happen after the assignment. Note that this is not a problem for approaches that use abstract objects as the points-to analysis takes care of assignments.

2.3.2.3 Aliasing

Two lvalues are *aliases* if they refer to the same object. This complicates things further because an assignment to an object's field accessed through one alias may change the object being referred to when an access involving the other one occurs. For example, in Figure 2.8(b) `me` and `khilan` are aliases. Consequently, `you.account`'s balance is being updated in the second line. Aliases are usually computed using a points-to analysis. However, if this information is not available, all we can do is be conservative and assume that `me`, `you` and `khilan` could all alias each other. This is because our lock inference analysis must be correct for all executions.

Autolocker [41] assumes that all non-global lvalues of the same type are aliases, while [6] treats the receivers of lvalues that have the same final field as possible aliases. For example, potential aliases in lvalues `x.f.g.s.g` and `q.g` are: `x.f`, `x.f.g.s` and `q`. Finally, the approach in [29] uses coarse locks when aliasing makes it unclear which objects are being accessed. [8] distinguishes between must- and may-aliases and uses this information to impose constraints

Figure 2.9: *Heap-centric view of iterating through a linked list.*



on the locks that protect them: lvalues that always alias each other can use per-instance locks, while lvalues that may alias each other must be protected by the same global lock.

2.3.2.4 Unbounded accesses

We revisit the linked list traversal example of Figure 2.7. As mentioned above, we cannot infer at compile-time how many nodes will be accessed, as each iteration of the while loop will access one node and we don't know how many times the while loop will iterate. To ensure our analysis is correct and covers all cases, we can only assume that this number is infinite. This is okay if we are inferring abstract objects because these are finite, but the lvalues approach generates an infinite set of lvalues! With respect to the start of the atomic section, the set of objects accessed would be {`n`, `n.next`, `n.next.next`, ...}.

Consider the linked list in Figure 2.9 to understand why. The diagram shows the node pointed to by `n` after each iteration. The key thing to note is that the object `n` points to after an iteration is `n.next` with respect to what it previously pointed to. To lock these accesses before the while loop, we want all lvalues to be in terms of what `n` points to there. This is the aforementioned set. But how do we represent such infinite sets? [5] caps the number of field lookups in lvalues, while [6] infer nondeterministic finite state automata, which are equivalent to regular expressions. For example, the set above can be written as `n(.next)*`.

2.3.2.5 Local/shared distinction

Accesses made inside an atomic section will typically consist of those objects that are not accessible by other threads (known as *thread-local*) as well as objects that are (known as *thread-shared*). Note that thread-local data do not need to be protected, as there is no contention for them. Hence, an optimisation employed by three approaches [41, 21, 29] is to ignore such thread-local accesses. With [41], it is implicit because they assign locks to data that could potentially be shared, whereas with the other two [21, 29], a static analysis is employed.

We might be able to further reduce the number of inferred accesses by noting that thread-locality may be too strong a requirement, particularly in a weakly atomic implementation, which lock inference approaches inherently are. Another approach [31] is to distinguish between accesses made inside atomics and accesses made outside. This means that if a datum is only accessd by one atomic, regardless of whether it is thread-local or thread-shared (it may be accessed concurrently outside the atomic), there is no need to protect it.

2.3.3 Inferring locks

Having inferred which shared accesses occur within atomic sections, the next step is to infer a set of locks that ensures they do not conflict with each other. There are a number of ways in which existing work differs here, including whether they first isolate conflicting atomic sections, how they map accesses to locks, whether they minimise the number of locks and the chosen granularity of locks. We now look at these areas.

2.3.3.1 Isolating conflicting atomic sections

[21] identify that existing lock inference work can be categorised as being either top-down or bottom-up. Top-down approaches [21, 65] first identify which atomic sections may conflict with each other and then infer a set of locks which ensures they do not execute in parallel, while at the same time allowing those that do not conflict to do so. Conflicting atomics are detected by finding intersecting read/write sets. [21] improve upon this by also considering which atomic ections could actually execute concurrently, using a refined *May-Happen-in-Parallel* analysis [45].

Bottom-up approaches [41, 29, 8, 6, 5] on the other hand, begin from the data accesses and then derive a set of locks from them. This could have the disadvantage of leading to more locks being inferred.

2.3.3.2 Mapping accesses to locks

In object-oriented languages, each object is typically protected by its own lock. However, in general the relationship between locks and data is flexible. Almost all lock inference work [29, 8, 65, 21, 6, 5] performs this mapping automatically, with the exception of [41], which allows the programmer to annotate what locks protect what data. This has the advantage that it gives developers more control over performance as they can control the granularity of lock. However, it adds the overhead of annotations and also relies on the programmer using them correctly.

2.3.3.3 Minimising the number of locks

A number of approaches also employ additional techniques to reduce the number of locks. [8] and [65] use 0-1 ILP and formulate lock inference as an optimisation problem. [65] also use heuristics, such as "all conflicting atomic sections must have one lock in common." [21] also use heuristics. [29] coalesce locks which are always acquired together.

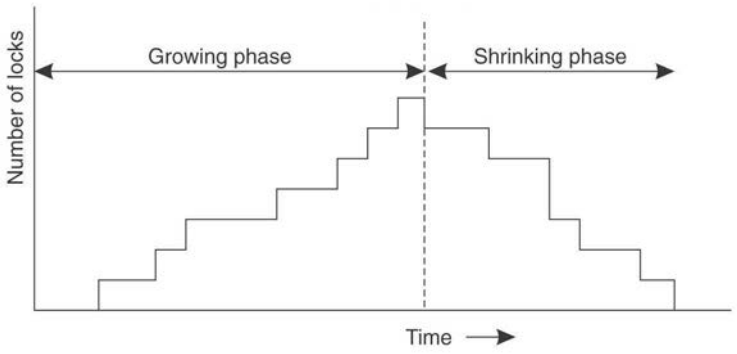
2.3.3.4 Locking granularity

The number of datum protected by a lock is known as the *locking granularity* and 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.

In approaches that use abstract objects [29], a lock is associated with each allocation site. 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 the same allocation statement and will consequently share the same lock.

Lvalues allow per-instance locks [41, 6, 5, 8], however, aliasing [8] and unbounded accesses [6, 5] often mean that coarser locks are used. A possible solution is to use locks of differing

Figure 2.10: *Locking policies that adhere to two-phase locking (2PL) will guarantee atomicity. Image adapted from http://rainbow.mimuw.edu.pl/SO/Wyklady-html/Tanenbaum/05-26.jpg.*



granularities at the same time, i.e. per-instance locks where possible and coarser locks for unbounded accesses. This is known as *multi-granularity* locking and is used by [6, 5].

Finally, top-down approaches to lock inference [65, 21] could be considered coarse, as a small set of locks protect a large number of accesses. However, their goal is to prevent conflicting atomic sections from running in parallel. Bottom-up approaches in conjunction with a suitable locking policy (see below), have the advantage that they can allow conflicting atomic sections to overlap, thus potentially allowing more concurrency.

2.3.4 Acquiring/releasing locks

Having inferred the locks to be acquired, the last step is to insert them into the program in the form of acquire and release operations. However, where they are inserted can have a huge impact on concurrency. Furthermore, the order in which locks are acquired can lead to deadlock. We look at these two issues here.

2.3.4.1 Locking policy

In this section we refer to a result from database theory [11]. However, it is important to note that atomicity has a different meaning there. We regard atomicity to mean 'in a single-step,' while in databases it means to execute completely or not at all [64]. These definitions are different. The former says that interleavings from threads do not affect the final result, while the latter says that execution does not stall half-way. The databases term for the semantics of atomic sections is *serialisability* [41]. All remaining uses of 'atomic' in this section should be read as 'serialisable.'

Database theory says that a sequence of `lock()` and `unlock()` operations (called a *locking policy*) will guarantee atomicity provided no locks are acquired after the first `unlock()`. This implies that in general, our inferred locking policy consist of a phase during which locks are acquired followed by a phase where locks are released. Such a restriction is called *two-phase locking* (2PL). The two phases are called *growing* and *shrinking* respectively. Figure 2.10 provides a graphical illustration.

A basic version, used by four approaches [29, 65, 21, 5], acquires all necessary locks at the start of the atomic section and releases them at the end. Although simple, it has the disadvantage of drastically impacting concurrency, especially when objects are required for a

short period of time and other atomic sections are waiting to access them. Additionally, atomic sections that require a large number of locks may have to wait a long time before they can start. In the worst case, they may never get to execute. To enable more parallelism, several variations of this basic 2PL policy exist [11]:

- **Late locking:** Delays acquiring a lock until absolutely necessary and releases them all at the end. For example, each lock is acquired just before the object it protects is accessed for the first time. The advantage is that atomic sections spend less time waiting to start. However, late locking is complicated by the ordering on locks for avoiding deadlock. In the worst case, the resulting policy can be the same as the basic one. This version is used by two approaches [41, 8].
- **Early unlocking:** Locks are acquired at the beginning of the atomic section, but are released when no longer required. This can achieve more parallelism than the basic policy, however it requires knowing when objects are no longer needed. This can be difficult at compile-time. It is used by [6].
- **Late locking and early unlocking:** 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. This policy can achieve more parallelism than the above two but it is complicated by their respective issues.

2.3.4.2 Deadlock

If two or more threads try to acquire the same locks but in different orders, it can lead to a state where they wait for each other called *deadlock*. Existing lock inference approaches can be divided into either dealing with deadlock at compile-time, which we shall denote a *static* approach, or at run-time, which we shall call a *dynamic* approach.

Static approaches can either avoid deadlock by ensuring locks are acquired in some globally defined order. When the number of locks is finite such as when using abstract objects, it is possible to determine this ordering. This is because all locks to be acquired are known.

However, when inferring lvalues, this isn't possible without being overly conservative. For example, [41] imposes an ordering on lvalues at compile-time by treating all lvalues with the same type as aliases. This has the side-effect that because of other dependencies on the locking order created by assignments and the fact that they use late locking, their approach can end up rejecting programs that they cannot guarantee will not deadlock. [8], who extend [41], also order lvals but use global locks when this is not possible. Other approaches which statically order are [29] and [65]. [21] uses static (i.e. compile-time) locks when deadlock is possible.

[6] and [5] differ from the aforementioned approaches in that they avoid deadlock dynamically. [6] maintains a waits-for graph. They acquire all locks at the start of the atomic section and when deadlock occurs, the atomic section which caused it releases all previously acquired locks and tries to acquire them again, essentially rolling back the locking phase. [5] on the other hand, ensure that all ancestors in the multi-grain lock hierarchy are already locked.

2.3.5 Additional features

We finally look at the additional features supported by some approaches.

2.3.5.1 Nested atomic sections

Almost all lock inference approaches merge nested atomic sections with their parent, creating one large atomic section. This can negatively hit concurrency. The exception to this is [21] which

treats a nested atomic section as distinct from its parent. This means that their locking policy is not two-phased, however, there is also the additional concern that the outermost atomic section is no longer atomic. This would be equivalent to open-nesting in the transactional memory world [43].

2.3.5.2 Condition variables

Condition variables allow a thread to block waiting for some condition to be true, and to be subsequently woken up when it is. The semantics of this inside atomic sections may be tricky because waiting for a condition to become true might require releasing other locks to allow queues to be modified for example. This could break atomicity. Conditional variables are supported by [41, 21, 65].

2.4 Soot

Soot [60] is a Java optimisation framework for analysing and transforming Java bytecode. It has four intermediate representations, the most commonly used of which, is *Jimple*. This is a typed 3-address code representation. Soot also contains a number of analyses already implemented within it, such as points-to [38, 39], as well as providing useful features like call-graph construction.

We will implement our analyses in Soot.

Chapter 3

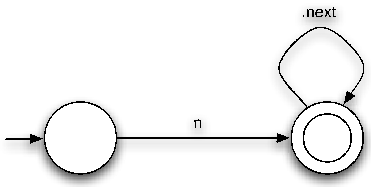
Progress

During my Masters [20], I worked with third year PhD student David Cunningham[1], on an efficient representation and analysis for inferring lvalues. The crux of our approach was to use non-deterministic finite state automata (NFA) to neatly handle unbounded accesses. We also looked at how to lock such accesses, detecting deadlock at run-time and how to support early unlocking in our system. This work was also published [6].

Figure 3.1 shows the linked list traversal example from Figure 2.7 together with the NFA inferred by our analysis. Note that NFAs are equivalent to regular expressions, hence in this example it is equivalent to `n(.next)*`.

Figure 3.1: *An example of our algorithm inferring object accesses inside a loop.*

```
Node n = list.head;
while(n != null)
    n = n.next;
```



An automaton is a finite-state machine, consisting of states, transitions and transition labels. Formally, it is a 5-tuple $M = (Q, \sum, \delta, i, F)$, where Q is a finite set of states, $i \in Q$ is the start state, $F \subseteq Q$ is a set of final/accepting states, $\sum$ is the alphabet of labels allowed on transitions and δ is a partial mapping $\delta : Q \times (\sum \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$ denoting transitions.

In our representation, transition labels can be either variable names, field names or [] representing an array access. For further details about the transfer functions, please see [6, 20].

3.1 Scaling the approach to Java

I began the PhD by implementing our analysis in Soot, extending the transfer functions accordingly to support static accesses and exceptions. However, given that our analysis is context-sensitive and uses call strings, I quickly ran into scalability problems when analysing a "hello world" atomic section, as shown in Figure 3.2.

This was due to the thousands of methods statically reachable from `PrintStream.println()`. I tried using the approach detailed in [34], however, the analysis still didn't scale. Consequently,

[1] http://www.doc.ic.ac.uk/~dc04/

Figure 3.2: *A "hello world" atomic section.*

```
atomic {
  System.out.println("Hello world");
}
```

Figure 3.3: *Our transformers.*

Statement	Transformer
$[x = y]^n$	$\{x \xrightarrow{\lambda l.l} y\}$
$[x = y.f]^n$	$\{\Lambda \xrightarrow{\lambda l.(0,n)} y\} \cup \{x \xrightarrow{\lambda l.(n,l.dest)} .f\}$
$[x.f = y]^n$	$\{\Lambda \xrightarrow{\lambda l.(0,n)} x\} \cup \{.f \xrightarrow{\lambda l.l} .f, .f \xrightarrow{\lambda l.(0,l.dest)} y\}$

due to this experience and the advice given by people I've met at conferences, I have switched to using a summary-based approach instead.

3.2 Summaries

In the Background chapter, we mentioned that IDE data flow analyses have a very efficient representation already defined for them in the literature [52], hence, we decided to reformulate our analysis in terms of the IDE framework.

In our original analysis, the data flow facts are automata. Hence, the first step was to convert them to environments; that is, mappings from some finite set of symbols D to elements of a finite semi-lattice L. We represent an automaton as a mapping from transition labels to pairs of the form (q_1, q_2) where $\{q_1, q_2\} \subseteq Q$. Let $StatePairs = Q \times Q$. Then, we choose $D = LocalVars \cup FieldNames \cup ClassNames \cup \{[]\}$ and $L = \mathcal{P}(StatePairs)$.

3.2.1 Transformers

The next and final step is to define the transformers, which describe how an environment is changed by a program statement. For full details about transformers, please refer to [52]. We modify the semantics for transformer edges $d_i \xrightarrow{\lambda l.l} d_j$ to mean "make all lattice elements mapped by d_i in the incoming environment to become lattice elements mapped by d_j in the outgoing environment, as described by the jump function on the edge." Furthermore, we assume implicit identity edges $d_i \xrightarrow{\lambda l.l} d_i$ for d_i that are not transformed. We represent kills as edges of the form $d_i \xrightarrow{\lambda l.l} \epsilon$ and to preserve kills when taking the transitive closure, we assume an implicit edge of the form $\epsilon \xrightarrow{\lambda l.l} \epsilon$. Finally, generated accesses are represented as $\Lambda \xrightarrow{\lambda l.(q_0,q_1)} d_i$, where $\{q_0, q_1\} \subseteq Q$ and to preserve them across transformer composition, we assume the implicit edge $\Lambda \xrightarrow{\lambda l.l} \Lambda$. Figure 3.3 gives our transformers, assuming the aforementioned modified semantics.

3.3 Next steps

I have implemented this analysis in Soot and am currently experimenting with small programs.

Chapter 4

Research Plan

The core of the PhD will look at how to improve upon existing lock inference techniques, enabling more concurrency than current approaches allow. However, I will then consider three areas which existing work has not yet looked at:

- Formally investigate whether the semantics given by a lock inference implementation matches the higher-level semantics expected by the programmer using atomic sections. In particular, we ask the question of whether they adhere to the memory model when accesses to an object occur both within and outside an atomic section.
- Supporting parallelism, such as the fork-join model, within the atomic section. The motivation for this would be to take advantage of multi-core hardware.
- A hybrid lock inference/transactional system, which may use dynamic feedback or heuristics to determine where locks or transactions would be more beneficial. Current approaches typically only execute existing monitors, such as those declared using `synchronized`, transactionally or using the associated lock. Many papers have already hinted that this seems to be a logical next step.

4.1 Time-scale

I intend to allocate my time amongst the above tasks as follows:

- **Jul '09 - Oct '09**: Improve upon existing lock inference techniques.
- **Nov '09 - Feb '10**: Formal semantics of lock inference implementations.
- **Mar '10 - Jun '10**: Parallelism inside atomic sections.
- **Jul '10 - Oct '10**: Hybrid lock inference/transactional memory system.
- **Internship**: I intend to do a 3-month intership, during which I will work on a topic related to my thesis. However, at present, I don't know what this topic will be.

During this time, I may come across additional important areas to look at, which is why I have not allocated all available time. Furthermore, although I will be writing up as I go along, I will also require time at the end for completing my thesis.

4.2 Deliverables

Throughout the PhD, I will build and incrementally update a tool, provisionally called *lockguard*. This tool will be used to get results on first small then large pieces of software with improvements being made to the tool if necessary.

Bibliography

[1] M. Abadi, A. Birrell, T. Harris, J. Hsieh, and M. Isard. Implementation and use of transactional memory with dynamic separation. In O. de Moor and M. I. Schwartzbach, editors, *CC*, volume 5501 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2009. 13

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In D. A. Reed and V. Sarkar, editors, *PPOPP*, pages 185–196. ACM, 2009. 13

[3] 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. 15

[4] C. Blundell, E. Lewis, and M. Martin. Deconstructing transactional semantics: The subtleties of atomicity. *Workshop on Duplicating, Deconstructing, and Debunking (WDDD), June*, 2005. 13

[5] S. Cherem, T. M. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 304–315. ACM, 2008. 11, 21, 22, 23, 24, 25, 26, 27

[6] D. Cunningham, K. Gudka, and S. Eisenbach. Keep off the grass: Locking the right path for atomicity. In L. J. Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2008. 11, 21, 22, 23, 24, 25, 26, 27, 29

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. *Proc. International Symposium on Distributed Computing*, 2006. 15

[8] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 291–296. ACM, 2007. 11, 21, 22, 23, 25, 27

[9] R. Ennals. Software transactional memory should not be obstruction-free. 2006. 15

[10] Everything2.com. Lock convoying, February 2006. Available online at http://everything2.com/index.pl?node_id=1786627 (accessed 25-December-2006). 9

[11] P. Felber and M. Reiter. Advanced concurrency control in java. *Concurrency and Computation: Practice and Experience*, 14(4):261–285, 2002. 10, 26, 27

[12] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004. 8

[13] 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. 8

[14] 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. 8

[15] 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. 8, 10

[16] Foldoc. Livelock, June 2007. Available online at http://foldoc.org/foldoc.cgi?livelock (accessed 19-June-2007). 9

[17] K. Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. 15

[18] K. Fraser and T. Harris. Concurrent programming without locks. *Submitted for publication*, 2004. 15

[19] S. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003. 8

[20] K. Gudka. Implementing atomic sections using lock inference. Master's thesis, Imperial College London, June 2007. 29

[21] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT*, pages 353–364. IEEE Computer Society, 2007. 11, 21, 22, 24, 25, 26, 27, 28

[22] T. Harris. Design choices for language-based transactions. *University of Cambridge Computer Laboratory Tech. Rep., Aug*, 2003. 14

[23] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005. 14

[24] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003. 14, 15

[25] 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. 14

[26] 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. 15

[27] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, pages 175–190, 2004. 8

[28] 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. 15

[29] 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. 11, 21, 22, 23, 24, 25, 26, 27

[30] B. Hindman and D. Grossman. Strong atomicity for java without virtual-machine support. 2006. 13, 14

[31] 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. 15, 24

[32] 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.htmlsubj1. 6, 9

[33] 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. 9

[34] U. P. Khedker and B. Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In L. J. Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2008. 19, 29

[35] T. Krazit. Intel pledges 80 cores in five years. CNET News, September 2006. Available online at http://news.cnet.com/2100-1006_3-6119618.html (accessed 4-June-2009). 5

[36] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007. 10, 13

[37] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993. 6, 9

[38] O. Lhotak and L. Hendren. Scaling java points-to analysis using spark. *Lecture Notes in Computer Science*, 2622:153–169, 2003. 28

[39] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):3:1–3:??, Sept. 2008. 28

[40] D. Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGOPS Operating Systems Review*, 11(2):128–137, 1977. 9, 10, 13

[41] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006. 11, 20, 21, 22, 23, 24, 25, 26, 27, 28

[42] M. Moir. Transparent support for wait-free transactions. *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997. 15

[43] J. Moss. Open nested transactions: Semantics and support. *Workshop on Memory Performance Issues*, 2006. 28

[44] A. Mycroft. Programming language design and analysis motivated by hardware evolution. In H. R. Nielson and G. Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2007. 5

[45] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In *SIGSOFT FSE*, pages 24–34, 1998. 25

[46] F. Nielson, H. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999. 15

[47] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996. 5

[48] D. J. Pearce. *Some directed graph algorithms and their application to pointer analysis*. PhD thesis, Imperial College of Science, Technology and Medicine, February 2005. 21

[49] K. Poulsen. Tracking the blackout bug. *Security Focus*, April 2004. Available online at http://www.securityfocus.com/news/8412. 6, 9

[50] 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. 14

[51] A. Rountev, M. Sharp, and G. Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In L. J. Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2008. 19

[52] Sagiv, Reps, and Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS: Theoretical Computer Science*, 167, 1996. 19, 30

[53] 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. 15

[54] 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. 15

[55] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 181–194, New York, NY, USA, 2008. ACM. 13

[56] M. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, 1987. 10

[57] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981. 19

[58] 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. 5

[59] C. Szydlowski. Multithreaded technology and multicore processors. *Dr. Dobb's Journal*, May 2005. Available online at http://www.ddj.com/dept/architect/184406074. 5

[60] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In S. A. MacKay and J. H. Johnson, editors, *CASCON*, page 13. IBM, 1999. 28

[61] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, 2006. 8

[62] A. Welc, A. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. *European Conference on Object-Oriented Programming*, 2006. 11

[63] Wikipedia. Lock convoy, December 2006. Available online at http://en.wikipedia.org/wiki/Lock_convoy (accessed 25-December-2006). 9

[64] Wikipedia. Acid, 2007. Available online at http://en.wikipedia.org/wiki/ACID (accessed 7-June-2007). 26

[65] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In J. N. Amaral, editor, *LCPC*, volume 5335 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008. 11, 22, 25, 26, 27, 28