University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Lock Inference for Java

Khilan Gudka

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, September 2012

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. Transactional memory (TM), is another automated technique for providing atomicity, but it imposes restrictions on the language features that can be used inside atomic blocks, or serializes all blocks that use such features. Therefore, from a language designer's point of view, the challenge is to implement atomic sections without compromising performance or expressivity.

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. We focus on lock inference techniques for tackling large Java programs making use of mature libraries. This improves upon existing work, which typically does not handle large libraries, or which provides conservative approximations that reduce analysis precision and may be unsound. Our approach allows us to handle atomic sections involving complicated real-world side effects, such as I/O, while still permitting atomic sections to run concurrently in cases where their lock sets are disjoint.

To validate and evaluate our claims, we have implemented our techniques in LOCKGUARD, a fully automatic tool that translates Java bytecode containing atomic sections to an equivalent program that uses locks instead. We show that our techniques scale well and we obtain performance comparable to the original hand-crafted lock-based implementations of our benchmarks.

Acknowledgements

I would like to express sincere gratitude to:

- Firstly, I would like to thank my supervisor Professor Susan Eisenbach for being such a great advisor and friend over the last few years. Her trust in my ability and patience have enabled me to gradually develop into a confident researcher. I would not have been able to reach such a successful conclusion to my thesis without her support and guidance.

- I would also like to thank my second supervisor, Professor Sophia Drossopoulou for our discussions and her enthusiasm in all aspects of my work.

- I would like to thank my Microsoft sponsor, Tim Harris, for being a great mentor and accepting me as part of the Microsoft PhD Scholarship program. I have enjoyed the many chats we have had about the work.

- Thanks also to David Cunningham, who first introduced me to atomic sections and lock inference. He also had the original idea of representing object accesses as path expressions. I feel fortunate to have been able to work with him during my Masters thesis and first year of the PhD. His helpful nature, ability to think very quickly and motto of "functionality first" inspired me in many ways.

- I would like to show gratitude to the SLURP research group for the immensely useful feedback for various aspects of my work and the interesting discussions over the years. In particular, I would like to thank my office mate Tristan for the many whiteboard sessions and suggestions about how to develop my ideas.

- I would like to deeply thank my parents for nurturing me into the person that I am today. They have constantly encouraged and supported me, both emotionally and financially, to achieve higher and reach my true potential. There is no way that I can repay them.

- I would finally like to thank my dear wife, Meha for always being there for me and constantly showering her love and support. Thank you for patiently tolerating the many

months that I spent writing this thesis. You are my best friend and life partner. I pray that may our love for each other only grow stronger with every passing moment.

Statement of Originality

The implementation of LOCKGUARD and the algorithms described in this thesis are my own work.

The original idea of representing object accesses as path expressions originates from David Cunningham. David Cunningham, Professor Susan Eisenbach and I co-authored a paper [CGE08] describing this approach and the idea of representing path expressions as non-deterministic finite automata. This is part of David's PhD, but we jointly came up with the idea during my master's thesis and the second year of his PhD. The idea to use type locks and multi-granularity locking to support the two granularities of locking were also David's idea.

Professor Susan Eisenbach, Tim Harris and I co-authored a paper, entitled "Lock Inference in the Presence of Large Libraries." This formed the technical basis for chapters 3, 4 and 5. The technical contributions of the paper are my own.

Professor Susan Eisenbach and I co-authored a paper, entitled "Fast Multi-Level Locks for Java." The technical contributions of this paper are also my own and form the basis for Section 5.2. The multi-granularity locking protocol is borrowed from Gray et al [GLP75]. Professor Sophia Drossopoulou provided detailed feedback about this paper.

In Chapter 4, the CFG summarisation technique is borrowed from Rountev et al [RSX08] and their paper gave us the inspiration to use the IDE analysis framework for our approach. All remaining analyses and techniques are my own work.

Professor Eisenbach has also proof read and contributed detailed suggestions throughout this thesis. Any mistakes remaining are my own.

‘In the joy of others, lies our own.’
Pramukh Swami Maharaj

Contents

List of Tables

List of Figures

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 [Myc07]. 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. Intel predicts that future processors will contain hundreds or even thousands of cores on a single chip [GC09].

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 [Sut05, Szy05]. That is, structured as a set of logical activities that execute simultaneously. For example, a concurrent 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 [Sut05], 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** [Ous96]. In the following sections, we

look at why this is the case.

1.2 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* [Ous96]. 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 same 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 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 [LT93, Jon97, Pou04].

```
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)

Figure 1.1: An example race condition that occurs when two threads T1 and T2 proceed to increment a counter at the same time without synchronisation.

1.2.1 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 each other.

```
class Counter {
  int counter = 0;

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

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

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

Figure 1.2 shows a race-free version of our counter example. The `synchronized` keyword is Java syntax that requires the invoking thread to first acquire an exclusive lock on the `Counter` 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.

1.2.2 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

```
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)

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

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`.

The former case whereby shared accesses are not protected is also referred to as *low-level data races* whereas the latter case of a related sequence of protected shared accesses not being atomic is also known as a *high-level data race* [AHB03]. When referring to race-freedom in this thesis, we refer to the low-level notion.

1.2.3 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 [FQ03b, FQ03a], type inference [FFL05], model checking [HRD04], theorem proving [FQ03c] and run-time analysis [FF04, WS06]. However, enforcing atomicity is still left to the programmer, usually using locks.

1.2.4 The ~~joys~~ complexities of locks

A lock is a data structure that can be in one of two states: *acquired* and *free*. It also has two operations `lock()` and `unlock()` allowing it to be acquired and released respectively. A thread acquires the lock associated with a shared object before accessing it. If the lock is held by another thread, it must wait until that thread releases it. In this way, threads are prevented from performing conflicting operations and interfering with each other.

The main problem with using locks is that they are imperative—the programmer is responsible for enforcing atomicity using them. In object-oriented languages, each object is typically protected by its own lock. However, in general the relationship between locks and objects is flexible. The number of objects protected by a lock is known as the *locking granularity*. This presents a tradeoff between simplicity and parallelism. A *coarse* granularity requires few locks, but permits less concurrency because threads are more likely to contend for the same lock.

Conversely, *fine grained* locks protect fewer objects resulting in a larger number of locks but allow more accesses to proceed in parallel. Some examples of locking granularities include:

- 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.
- Mutual exclusion lock per 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.
- 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.

Each of the above present trade-offs in terms of performance and complexity that the programmer has to choose from. Programmers aim to get the best performance out of their software. However, the complexity of concurrency control can increase dramatically with the number of locks:

- Forgetting to acquire a lock re-invites the problem of interference (safety violation).
- Acquiring locks in the wrong order can lead to deadlock (progress violation).

Figure 1.4(a) 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 [Art01]).

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 may result in 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 it be prevented? You would need to ensure that `c1.equals(c2)` and `c2.equals(c1)` were 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) [KB02].

- **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

```
class Counter {
  int counter = 0;

  synchronized void increment() { ... }

  synchronized void reset(int val) { ... }

  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: 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.

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 [com06, Wik06].

- **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 they are too low-level and leave the onus on the programmer to enforce safety and liveness properties. This is not just felt by novice programmers, As a result, even experts can end up making

mistakes [HP04]. The worst part is that these problems are hard to detect at compile-time and their impact at run-time can be disastrous [Jon97, LT93, Pou04].

1.2.5 What about lock-free programming?

Lock-free programming [Fra03] 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 `java.util.concurrent` framework [Lea05], introduced in Java 5, 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 [HSY04, FR04]. Thus, such a methodology doesn't seem like a practical solution in the short run.

1.2.6 Intractability of programmer-enforced atomicity

In addition to the problems that arise when trying to enforce atomicity using locks, it actually may not 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 for atomicity (if the method accesses other objects via instance fields, 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.

1.3 The quest for better abstractions

Given that programmers face an inevitable turn towards concurrency and the problems associated with current abstractions, a lot of research is currently being done to find ways to make concurrent programming easier and 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 these languages should look like, they suggest this shift should be gradual [Sut05].

Many have proposed race-free variants of popular languages that perform type checking or type inference to detect if a program contains races [Boy04, CDE, Gro03], while others abstract concurrency into the compiler enabling programmers to specify declaratively their concurrency requirements through compiler directives [vos03]. Alternative models of concurrent computation have been suggested such as *actors* [Agh86] and *chords* [BCF04] 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 [BGK+06, Hos06].

However, these proposals either require programmers to dramatically change 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 Atomic sections

The difficulty of manually enforcing atomicity has led researchers to consider a language-level abstraction to do the job instead. *Atomic sections* [Lom77] 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. Figure 1.5 shows an implementation of our double-counter-reset example using atomic sections (denoted using the `atomic` keyword).

Unlike locks, they are declarative and thus relieve the programmer from the complexities associated with concurrency control. They enable them to think in terms of single-threaded semantics, also removing 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 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 automatically. 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, there is no longer the potential for deadlock to occur as the underlying implementation ensures that safety and progress violations do not occur.

1.4.1 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 [Lom77], whereby interrupts are disabled while a thread executes inside an atomic section.

```
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)

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

- **Co-operative scheduling:** Involves intelligently scheduling threads such that their interleavings ensure atomicity [Sco87].

- **Object proxying:** A very limited technique whereby proxy objects are used to perform lock acquisitions before object invocations at runtime [FR02].

- **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 [HLR10].

- **Lock inference:** A compile-time 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 [HFP06, MZGB06, CCG08, HPV07, CGE08, EFJM07, ZSZ+08].

- **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 [WHJ06].

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, most notably being poor support for irreversible operations such as I/O and system calls. Other drawbacks include high run-time overheads in both contended and uncontended cases and a large amount of wasted computation.

1.5 Lock Inference

Lock inference is a promising alternative: 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.

We feel that transactional memory's inability to support I/O and system calls is a significant disability, and is the reason why we are pursuing an implementation using lock inference instead.

1.6 Lock Inference for Java

Lock inference has a number of advantages over transactional memory, but in order for it to be useable, it is necessary to be able to apply it to languages that programmers currently use. However, prior lock inference work has paid little to no attention to this.

Programming languages typically come with a rich set of libraries that provide common functionality, such as maintaining a hashtable or performing I/O. However, libraries create a scalability challenge for static analysis [RSX08] because they are large and have a high cyclomatic complexity.[1] This leads to very long analysis times and lots of imprecision in analysis results.

[1] Cyclomatic complexity [McC76] is a measure of the number of linearly independent paths. Library call chains can be long and consist of large strongly connected components.

Although for libraries the issue of long analysis times is not important, as the results would only be computed once, actually being able to analyse the library and reducing the imprecision that the library introduces *are* important problems. The former determines whether such an analysis is even possible and the latter will have an impact on what locks are inferred and thus the resulting performance of the instrumented program. These are significant challenges for lock inference approaches because most real programs make extensive use of libraries. For example, consider a "Hello World!" program written in Java extended with atomic sections:

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

Lock inference prides itself on being able to support I/O, so one would expect it to be able to handle this library call. In practice, this example is non-trivial with a compile-time call graph containing 1150 library methods (for GNU Classpath 0.97.2) as shown in Figure 1.6. Analysing the library is a hard problem as is evident from the fact that existing work either ignores libraries [HFP06, CCG08, EFJM07, ZSZ+08], requires library implementers to annotate which method parameters should be locked prior to the call [MZGB06] or only considers accesses performed upto one level deep in library call chains [HPV07]. All of these have the potential that some shared accesses performed within the library may go unprotected, leading to atomicity violations.

Inspecting the callgraph for Hello World reveals that these methods come from `println(s)`'s call to `s.getBytes(encoding)`, which converts the string `s` to an array of bytes according to the given character set `encoding`. It does this by delegating to the class implementing the corresponding character set. However, it is this delegation that leads to the huge number of methods.

Character sets are provided by one or more `CharsetProvider`s. Two default providers are readily available that supply the most commonly used character sets (e.g. UTF-8). Third-party providers can also be loaded from the classpath. First, the two default providers are queried for the required character set. This results in lazy instantiation and initialisation of the providers

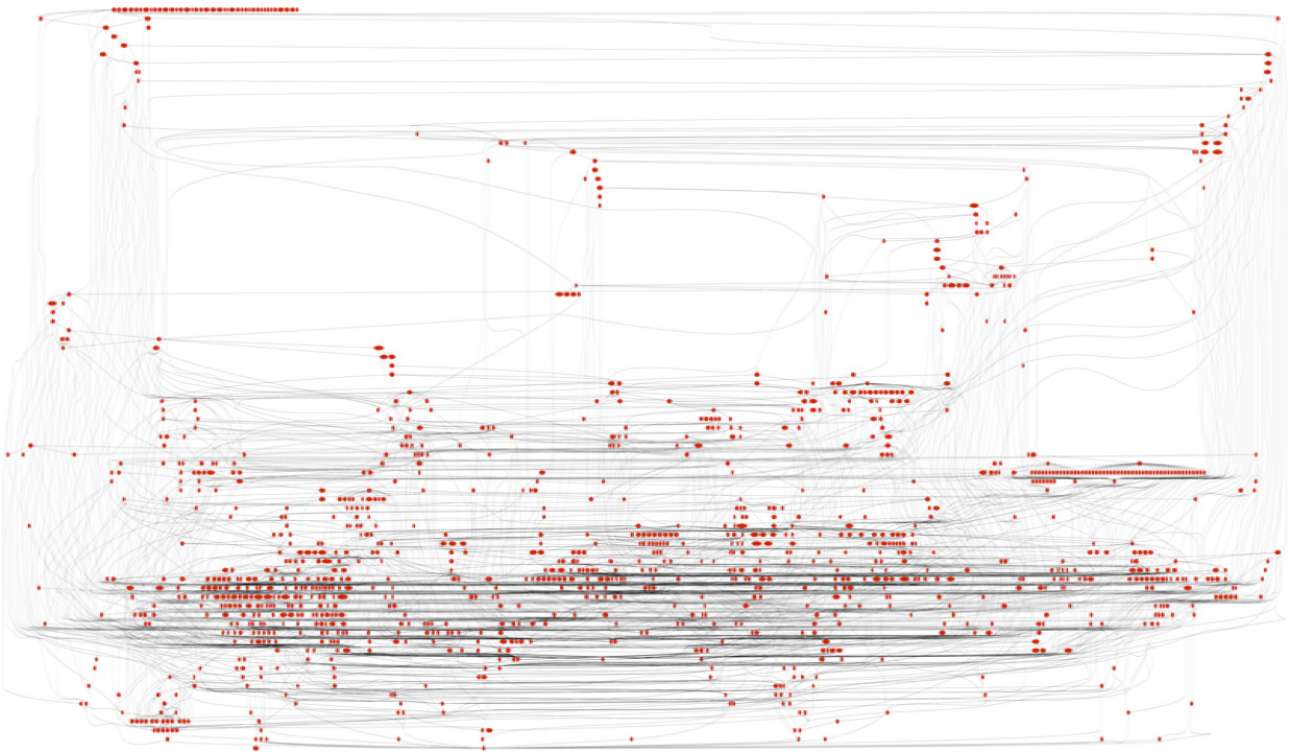


Figure 1.6: A callgraph for the hello world example, containing 1150methods. Each method is represented with a red circle.

(and instantiation of their combined total of 99 character sets plus storage of these instances in their respective `HashMaps`). Creating a provider is a privileged action, so it is performed through the `AccessController`. This involves saving the current security context, creating a new one, running the privileged action in this new context and then restoring the saved security context once finished. If the character set isn't found, third-party `CharsetProviders` are loaded reflectively from the classpath. This involves finding all resource files containing lists of `CharsetProviders`, iterating through each line of each file and loading each one. Loading a `CharsetProvider` is also a privileged action that must therefore be performed through the security framework. These loaded `CharsetProviders` are then individually queried for the necessary character set.

Once the string has been encoded, the returned byte array is written to `PrintStream`'s underlying `BufferedOutputStream`, followed by the bytes for the line separator. Finally, the entire buffer is flushed to *standard output*. If this I/O operation is interrupted, an `InterruptedIOException` is thrown, resulting in the current thread being interrupted by setting its interrupt status bit. Modifying the status of a thread is a privileged action.

Most of the large number of object accesses just described are performed only under special circumstances, such as when loading character sets or interrupting a thread. Unfortunately, lock inference is a static technique and must therefore conservatively ensure that all possible execution paths are protected.

This is perhaps the simplest program we would expect lock inference to be able to handle, but even the path inference analysis David Cunningham and I had previously developed [CGE08, Cun10, Gud07] was not able to scale to it. What this tells us is that special techniques need to be developed to tackle:

- **Complexity** - analyses need to be able to scale to the large code base and cyclomatic complexity of libraries.
- **Precision** - due to their widespread use, many common code paths and large numbers of unexecuted code paths, libraries can introduce many more locks than are required. Techniques are needed to reduce this number.
- **Performance** - the resulting performance should be comparable to that of manual locking. If this is not the case, then lock inference will not be seen as desirable.

1.7 Contributions

The thesis we argue is the following:

"It is possible to develop lock inference techniques that scale to real-world Java programs that make use of the library and still obtain performance comparable to hand-crafted locking."

The contribution of this thesis is a set of techniques that achieve the above. In particular, we present:

- An object access inference analysis for inferring which Java objects are accessed from within an atomic section, based on Sagiv's IDE framework [SRH96] (Chapter 3).

- A set of analysis optimisations for this access inference analysis that enable it to scale to the Java standard class library (Chapter 4).
- A set of lock optimisations to reduce the number of locks inferred, based on instance-local objects, class-local objects, method-local objects and locks dominated by other locks (Chapter 5).
- An implementation of all our analyses in the SOOT bytecode optimisation framework (Chapter 3, Chapter 4, Chapter 5).
- An extensive evaluation of our approach on real-world Java programs and libraries. We obtain performance of 3.5x to that of hand-crafted locking (Chapter 3, Chapter 4, Chapter 5).

1.8 Publications

During the PhD, I have published the following papers:

- **Lock Inference in the Presence of Large Libraries**
  Khilan Gudka, Tim Harris, Susan Eisenbach
  European Conference on Object-Oriented Programming 2012
- **Fast Multi-Level Locks for Java**
  Khilan Gudka, Susan Eisenbach
  EC^2 2010: Workshop on Exploiting Concurrency Efficiently and Correctly
- **Keep Off the Grass: Locking the Right Path for Atomicity**
  David Cunningham, Khilan Gudka, Susan Eisenbach
  Compiler Construction 2008

Chapter 2

Background

Before delving into the technical contributions of this thesis, we first visit some background areas to set the scene for our work. In particular, we look into the history of atomic sections, implementation techniques, relevant concepts from program analysis and survey prior lock inference approaches.

2.1 Atomic Sections

Atomic sections were first proposed by Lomet in his 1977 paper [Lom77]. However, they have only recently come into the forefront of programming language research. The last 10 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. We begin by looking more closely at the semantics of atomic sections.

2.1.1 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 [LR07, BLM05]:

- **Strong isolation:** 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 isolation:** atomicity is only guaranteed with respect to other atomic sections.

Ideally, atomic sections should provide strong isolation, 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 [AHM09, HG, SMSAT08], with [AHM09] reporting performance within 25% of an implementation only guaranteeing weak isolation.

It should be noted that providing strong isolation doesn't mean that an implementation has to directly support it. In fact, an implementation may only provide weak isolation 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 [ABH+09] has looked at a dynamic approach which verifies, at run-time, that atomic accesses of shared sdata 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 isolation 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.

2.1.2 Serialisability and two-phase locking

When we say that all atomic sections appear to have occurred in 'one step,' what this really means is that any concurrent execution involving them should be *serialisable*. Serialisability is a correctness condition from the database community [ram03] that (when adapted for atomic sections) states:

A concurrent execution involving atomic sections is serialisable, if it is equivalent to an execution in which all atomic sections are executed in some serial order.

Atomic sections can interleave their execution with other operations provided that the resulting concurrent execution preserves the above condition. In the case of strong isolation, this would additionally mean ensuring serialisability with respect to all shared accesses that are not inside atomic sections.

For transactional memory, serialisability is achieved by buffering updates. In lock inference, it is done by following the two-phase locking protocol (2PL). 2PL also originates from the database community and it dictates a restriction on the locking policy that guarantees a serialisable execution. This restriction is that no `lock()` operation should be performed once an `unlock()` has been performed. As a result, the program will consist of two locking phases: a *growing phase* during which locks are acquired, followed by a *shrinking phase* during which locks are released. Figure 2.1 shows a visualisation of these phases.

A simple example would be a basic policy that acquires all necessary locks at the start of the atomic section and releases them at the end. However, this may drastically impact 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

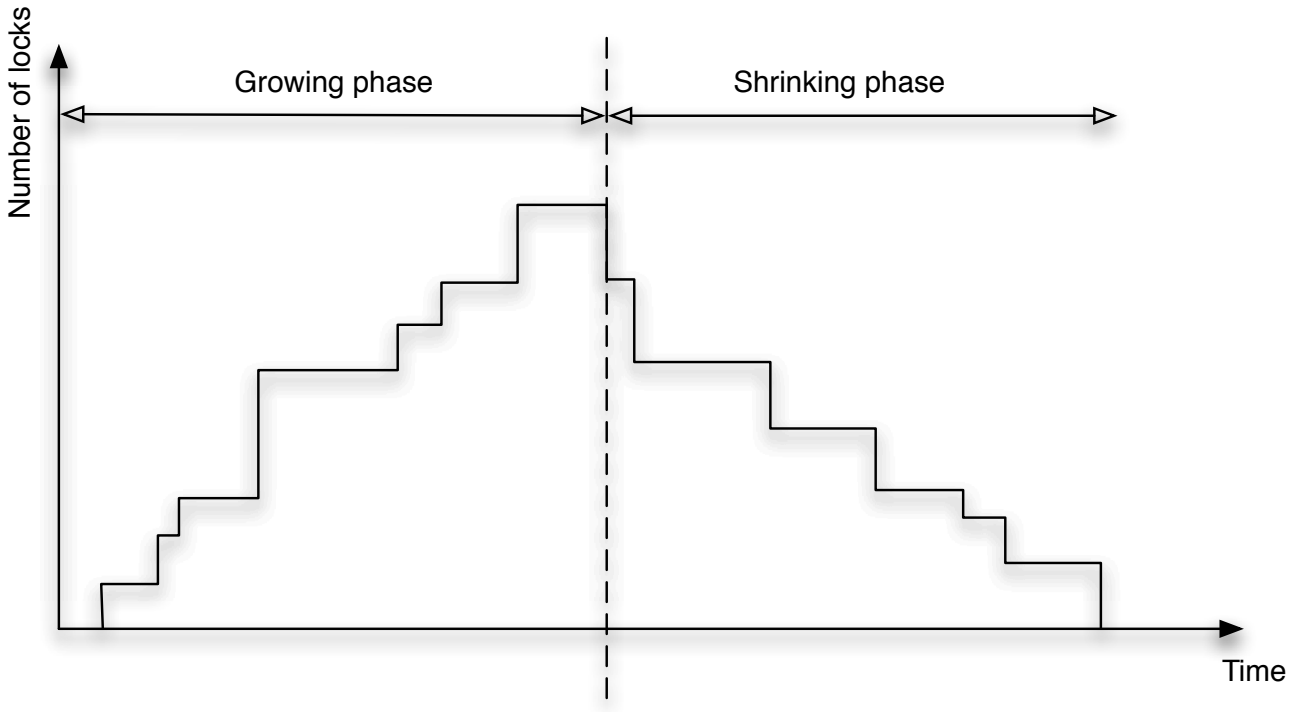


Figure 2.1: Locking policies that adhere to two-phase locking will guarantee serialisability.

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 policy exist [FR02]:

- **Late locking (or strict 2PL):** 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.

- **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.

- **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.

2.1.3 Atomic section nesting: flat, closed or open nesting

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 nesting semantics have been developed [Mos06][1]:

- **Flat nesting:** Any nesting structure is flattened so that all nested atomic sections are part of the outermost section.
- **Closed nesting:** 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 (i.e. no other thread has performed conflicting updates to the locations it has accessed), then its 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 nesting:** 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

[1] They have been proposed in the context of transactional memory but could also apply to atomic sections more generally

other threads may see inconsistent state if an outer transaction later aborts, requiring mechanisms such as locking [Mos06] 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.

For lock inference, as there is no buffering of updates, there is no distinction between flat and closed nesting. A lock can be released once we are sure that the associated shared data will no longer be accessed or that no other lock will be acquired (two phase locking requirement above). Consequently, it is possible for lock inference to have open nesting-like semantics, although all prior approaches and ours presented in this thesis assume flat nesting (i.e. locks are either all acquired at the start of the atomic section or as and when required, but are always released at the end of the outermost section).

2.2 Transactional memory

Transactional memory is the most popular way to implement atomic sections. It provides the abstraction of database-style transactions [EGLT76] 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.[2] 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 [Enn06], 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

[2]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.

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 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 [Har03]. 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 [Har05] and contention management [SIS05], but no general solution exists yet.

In comparison, lock inference 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 [HEM93], 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* [KCH+06]. 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 [AAK+05] and also into a thread's virtual address space [AAK+05, RHL05, MHW05]. However, as these data structures have to be traversed in hardware, the result is a more complicated HTM.

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. Nevertheless, things are on the turn with Intel announcing that its upcoming Haswell processor will contain hardware transactional memory support [Rei12].

The other problem is *portability*. Early proposals imposed architectural-dependent limitations; however, new hybrid approaches [KCH+06] 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 inference given that the latter doesn't use transactions, although a hybrid or the lock 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 [ST95] proposed a software-variant called software transactional memory (STM). 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 [HEM93, ST95]. Consequently, Shavit and Touitou's initial STM and many other early implementations [Fra03, FH04, HLMSI03, Moi97] 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 [Enn06]. Hence, many newer STMs have omitted the non-blocking requirement and instead use a combination of optimistic synchronisation and locks [DSS06, Enn06, HPST06] or only locks [HG06, SATH+06] (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 [HLMSI03, AR05, HPST06], better support for nested transactions [MH05], customisable contention management [GHKP05, HLMSI03, SIS05], conflict-driven notification [HMPJH05, CMC+06] and improved support for I/O and exceptions [Har05, Har03].

Even though there have been many advancements, the main focus has been on improving performance [HPST06]. 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.

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* [ST95, HF03, HMPJH05], 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* [Moi97, FH04, Fra03, HLMSI03, AR05, HG06, HPST06] 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 higher 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 object-based languages.

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, 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 [AR05].

Given that object-based STMs have lower 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 [Fra03, HLMSI03, Moi97], although other techniques

```
class Counter {
  int counter = 0;

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

Figure 2.2: A non-blocking implementation of the `Counter` class of Figure 1.2.

such as maintaining lists of reading and writing transactions in each object have also been proposed [AR05].

Non-blocking STMs

As already mentioned, 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 [FH04]. 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 1.2 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) and returns true, otherwise it does nothing and returns false. In this way, it *tries* to ensure that the update is atomic.[3]

Non-blocking algorithms can be classified according to the kind of progress guarantee they provide [FH04]:

- **Obstruction-freedom:** This is the weakest form of progress assurance: a thread is

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

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 obstructing it. The chance of this occurring is reduced using a contention manager, which determines what to do when contention for memory is detected. Example policies include exponential back off and aborting the conflicting transaction [HLMSI03]. 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 [SIS05].

- **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 [FH04, Fra03].

- **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 [FH04]. Kogan et al [kog12] propose a methodology to improve their performance by creating hybrid data structures that use a lock-free version most of the time, only reverting to a wait-free version when things go wrong. They call this technique *fast-path-slow-path*.

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 [Fra03], however this can be costly in terms of performance [ST95]. 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 [ST95], 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) [FH04], 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 [Enn06].

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 [Moi97]. 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 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 [MSS04]. Other significant disadvantages include wasted computation performed by a transaction that is destined to abort. In STMs that only perform validation just before committing [HLMSI03, HMPJH05], 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 [MSS04].

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

(a)

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

(b)

Figure 2.3: Example of opening an object before accessing it in object-based STMs [HLMSI03]. 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.

More recent non-blocking STMs include Fraser's FSTM [Fra03, FH04] and Herlihy et al's DSTM [HLMSI03, HLM06]. 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 [MSS04]. Consequently, validation has to be performed on each open for reading, which is extremely expensive and is thus a significant problem with optimistic approaches [MSS04]. 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 [MSS04].

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

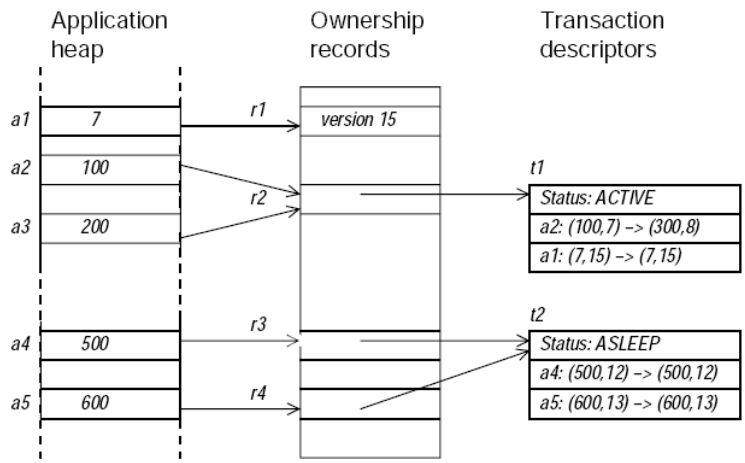


Figure 2.4: Data structures in Harris and Fraser's word-based STM [HF03].

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 [HMPJH05, HF03].

Harris and Fraser proposed an obstruction-free word-based STM [HF03] 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 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 [HF03].

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 [HF03] for suggested improvements).

One interesting feature though of this paper is that the programmer can specify an 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.

Omitting the non-blocking requirement

Semantically, non-blocking programs are desirable because they provide strong 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 [HF03, HLMSI03, HLM03].

However, recent work suggests that even this weakest guarantee is a hindrance [Enn06]. The main arguments for non-blocking STMs in the literature, aside from STMs originally being designed for use in non-blocking programs, include [Enn06]:

- **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. Also, STMs do not support I/O.

- **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 [Enn06], 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 [Enn06, SATH+06, HG06, DSS06] 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 [HG06]. 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 [SATH+06]. 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 inference 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 [Enn06, SATH+06] or only when required by another transaction (the holding transaction

is first given a chance to complete after which it is aborted) [HG06]. Lock inference 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 [HG06] 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.

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 [HMPJH05].

However, one significant hurdle it faces is expressiveness, given that atomic sections may contain operations that cannot be reversed. Buffering is one proposed solution [Har05, HG], 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 [HMPJH05], while some throw exceptions [RG05], although these are not practical in general. Irrevocable transactions [wel08] are a recent technique that enable irrevocable actions in transactions. When an operation such as I/O is encountered, the transaction transitions to an irrevocable state in which it will no longer roll back as a result of an external action performed by a different transaction. As a result, the system will guarantee that its subsequent actions (including, for example, I/O and system calls) will never be revoked and that its commit operation will succeed. However, only one irrevocable transaction is supported at once and roll back of revocable transactions still occurs.

Another major problem of STM is the significant overhead encountered 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 [Enn06], the introduction of object-based STMs [Moi97] and the ability to customise contention management policies [HLMSI03, SIS05, wel08]. 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 thesis employs lock inference 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 Inference

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 [HMPJH05] this is enforced using the type system, however, this isn't practical in more general languages such as Java. Alternative solutions include buffering [Har05], mutual exclusion locks [WHJ06] and irrevocability [wel08].

- **Performance overhead:** STM incurs significant overheads due to logging, validation and committing. In more recent STMs that use locks [HG06], 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 [HG06], 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. Although Intel will provide hardware transactional memory support in their upcoming Haswell processor [Rei12], it is unclear whether such hardware support will become widespread in commodity processors.

These limitations exist because transactional memory requires the ability to 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, currently 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

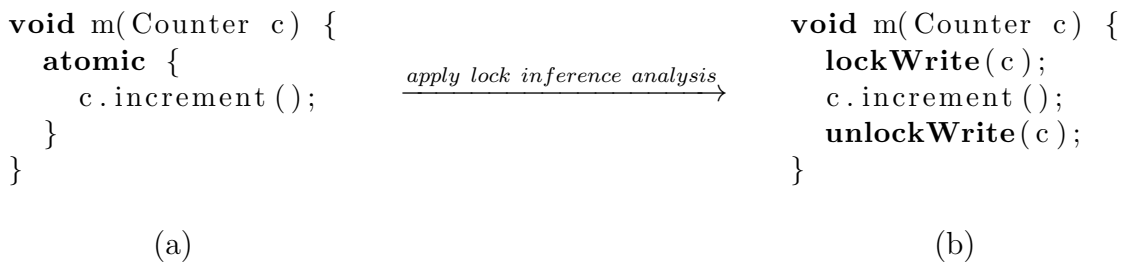


Figure 2.5: Lock inference example that uses reader/writer locks. (a) is the original program with atomic sections and (b) is the transformed version after applying the lock inference analysis. The analysis identifies which objects are accessed and maps them to the locks needed to protect them.

completely different approach to atomic sections that takes a preventative approach by using locks but with little or no effort from the programmer.

Lock inference [MZGB06] 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 [HG06] that also use locks, because lock inference ensures 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 2.13 shows an example of how lock inference works.

This approach 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 lot of work having been done in optimisations for it [BKMS98, ADG+99]. In some cases, this can be as cheap as setting/clearing a bit [WHJ06].

- **Lower 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* [Goe05] 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 inference 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 atomicity violations.

Before looking into the issues that must be taken into consideration to ensure a lock inference analysis meets these requirements and how existing work in this area has approached them, we briefly visit program analysis to introduce concepts that will be needed to understand and appreciate how lock inference works.

2.4 Program analysis

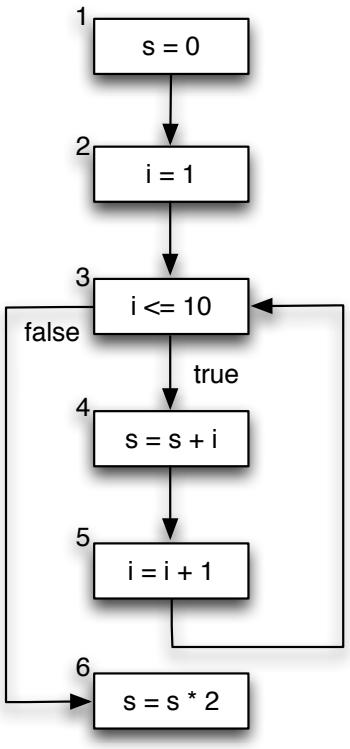
Lock inference relies heavily on program analysis to infer the locks necessary for atomic execution. Program analysis is used by lock inference to infer what objects are being accessed in atomic sections and what their corresponding locks are. It allows us to approximate run-time behaviours of programs at compile-time. This section provides a very brief overview of relevant concepts. For a detailed account, please refer to [NNH99, KSK09].

2.4.1 Data flow analysis

The approach to program analysis that is of relevance to this thesis 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 2.6(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

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

(a)



(b)

Figure 2.6: (a) is a program that calculates double the sum of 1 to 10. (b) is its control flow graph.

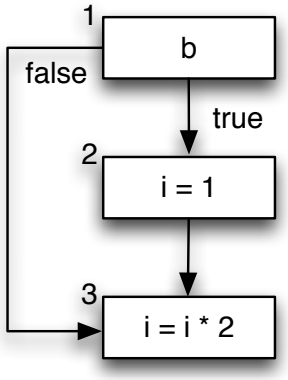
and false respectively. At compile-time, we typically cannot determine exactly which edges will be followed, therefore we must consider both of them. 'If' statements are similar.

In a nutshell, data flow analysis works by pushing sets of 'learned facts' 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 2.6(b), the incoming data are first combined using set union or intersection depending on the type of analysis. This combining of information from predecessors is called *taking the meet*.

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.

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

(a)



(b)

Figure 2.7: Simple program to demonstrate the difference between may and must analyses.

(a) May versus must analysis

Consider the example given in Figure 2.7 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`. 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 would union the result of each path. This is called a *may analysis*. In this, data from immediate predecessors are combined using set union.

(b) Forwards versus backwards analysis

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 2.7 requires looking at paths reaching it. On the other hand, determining if the value of `i` assigned at 2 is ever used requires looking at paths leaving it. 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
- Backwards, must
- Backwards, may

Entry and exit information

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.

Calculating the fixed-point

To calculate the final entry and exit sets, an iterative algorithm is used. Figure 2.8 gives a simple version of it in pseudo-code for a forwards analysis.

Here we use apostrophe ($'$) 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:

```
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)); } }
```

Figure 2.8: Simple iterative algorithm for computing the entry and exit sets of a forwards analysis.

```
worklist = all cfg nodes
while(worklist not empty) {
    n = pop next node off the worklist
    // 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));

      if (exit'(n) != exit(n))
        push n's successors onto the worklist } }
```

Figure 2.9: Worklist algorithm for computing the entry and exit sets of a forwards analysis.

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

The algorithm terminates when no entry and exit set changes between iterations. This is referred to as having reached a *fixed point*. Most algorithms for computing the fixed point use a worklist. This is a list of nodes whose entry and exit information need to be recalculated because the entry or exit information of at least one of their predecessors has changed. Figure 2.9 shows a pseudo-code version of the worklist algorithm.

2.4.2 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 2.10 gives a graphical description. Here, d is the entry information for n and d' is the data flowing back from m.

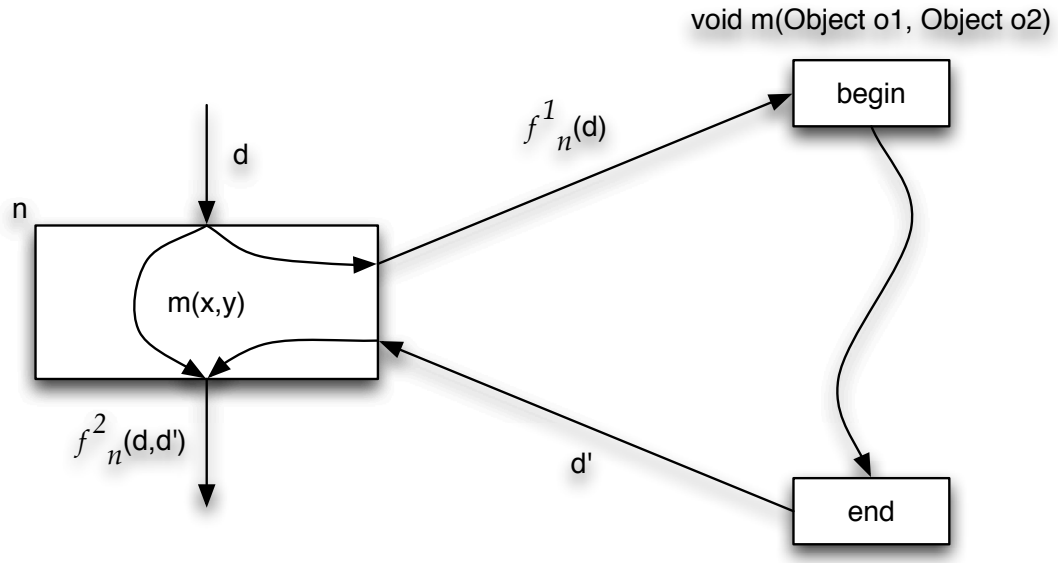


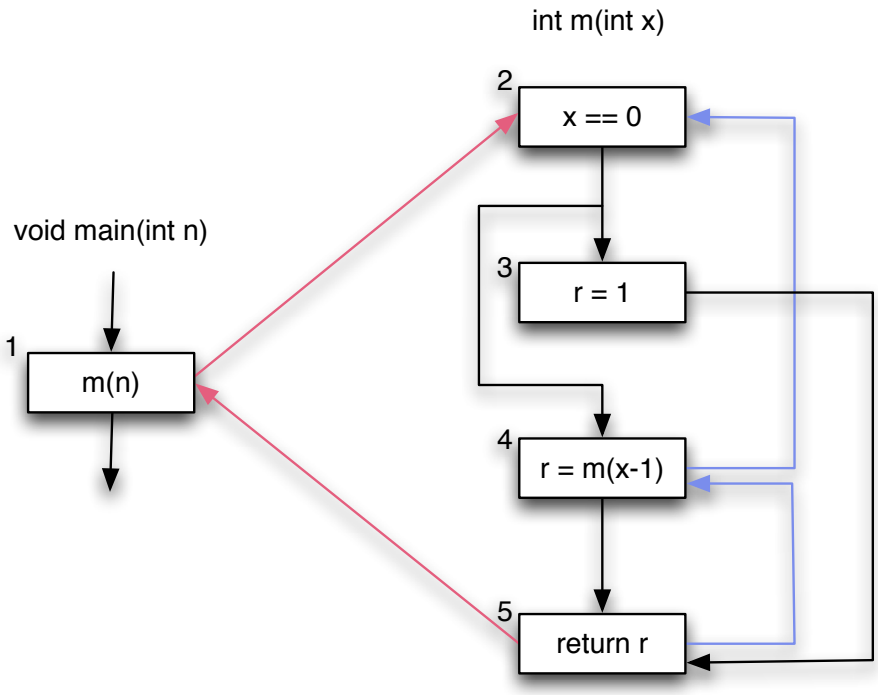
Figure 2.10: Interprocedural analysis.

Interprocedural analysis introduces two new functions $f^1_n(d)$ and $f^2_n(d, d')$. f^1_n 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^2_n 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

```
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)

Figure 2.11: Problem of valid paths.

parameters. The entry information may also be modified based on the returning data.

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 2.11. 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 2.11 there is nothing stopping the analysis from considering the path $1 \to 2 \to 4 \to 2 \to 3 \to 5 \to 1$ 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* [SP81]. 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*. Recording call contexts adds to the memory and computation overhead of the analysis and thus is typically avoided. Furthermore, context-sensitive often does not give a proportionate improvement in analysis precision. We tried implementing context-sensitivity in our initial analysis [CGE08], however found that it did not scale. Khedker et al [KK08] have found that by putting contexts into equivalence classes based on the data flow information they map to, their overhead can be reduced. However, we found that for our analysis it still did not scale.

Summaries

One of the widely known problems with using call strings [KK08], is that for programs with deep call chains, the number of 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 for interprocedural analysis, is the *method summary* approach [SP81] 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. Although calling contexts are not used, summaries are computed bottom-up from the callgraph and thus the problem of valid paths is largely avoided[4].

One of the main challenges of scalable summary computation is to find a representation that affords fast composition and meet operations. In the next subsection, we describe one such representation.

Interprocedural Distributive Environment (IDE) analyses

An important category of data flow analyes are the 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 [RSX08].

In an IDE problem, data flow values are called *environments*. That is, they are mappings of the form $D \to L$ 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 $L = \{\top, \bot\} \cup \mathbb{Z}$. Transfer functions describe how statements transform environments and are called *environment transformers*. These transformers have the special property that they are *distributive*. That is, $\forall t \in Env(D, L) \to Env(D, L)\ .\ \forall e_1, e_2 \in Env(D, L)\ .\ t(e_1 \sqcap e_2) = t(e_1) \sqcap t(e_2)$.

The scalability of a summary-based analysis depends upon the representation of transfer functions and how efficiently their composition and meet can be computed. For IDE Analyses, Sagiv et al [SRH96] show that transformers can be represented as compact graphs, called *pointwise*

[4]As we shall see later, the problem of invalid paths is not completely eliminated because summaries for methods that are part of the same strongly connected component must be computed together and can therefore suffer from data being propagated around invalid paths.

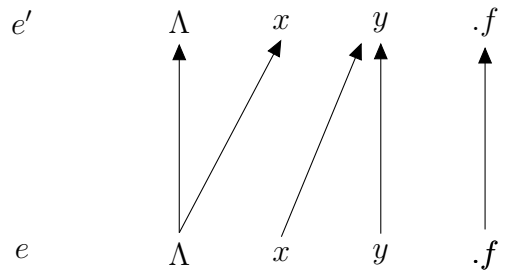


Figure 2.12: Pointwise representations for Figure 3.6 key transformers

representations, whose composition is the transitive closure and meet is essentially graph union or intersection. They represent a transformer $t : Env(D, L) \to Env(D, L)$, as a balanced bipartite directed graph $G_t = (D_1, D_2, E)$ where $D_1 = D_2 = D \cup \{\Lambda\}$ and E is a set of directed edges from nodes in D_1 to nodes in D_2.

Informally, these graphs describe how the exit environment e' is derived from the entry environment e. An edge $d_1 \xrightarrow{f} d_2$ in the graph means that $e'(d_2)$ is obtained from $e(d_1)$, with edge function $f : L \to L$ describing exactly how so. In the simplest case, $f = \lambda l.l$ (the identity function), so $e'(d_2) = e(d_1)$. If $e'(d_2)$ is dependent on multiple $e(d_k)$, the meet of the values (after applying the edge functions) is taken. New values (not derived from e) are introduced using the special symbol Λ. Figure 2.12 shows an example pointwise representation. In this particular case, $e'(x)$ is a new value, $e'(y)$ is derived from $e(y)$ and $e(x)$ and $.f$'s value is unchanged from e to e'. We will revisit IDE analyses and these pointwise representations in Chapter 3.

2.5 Literature review

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

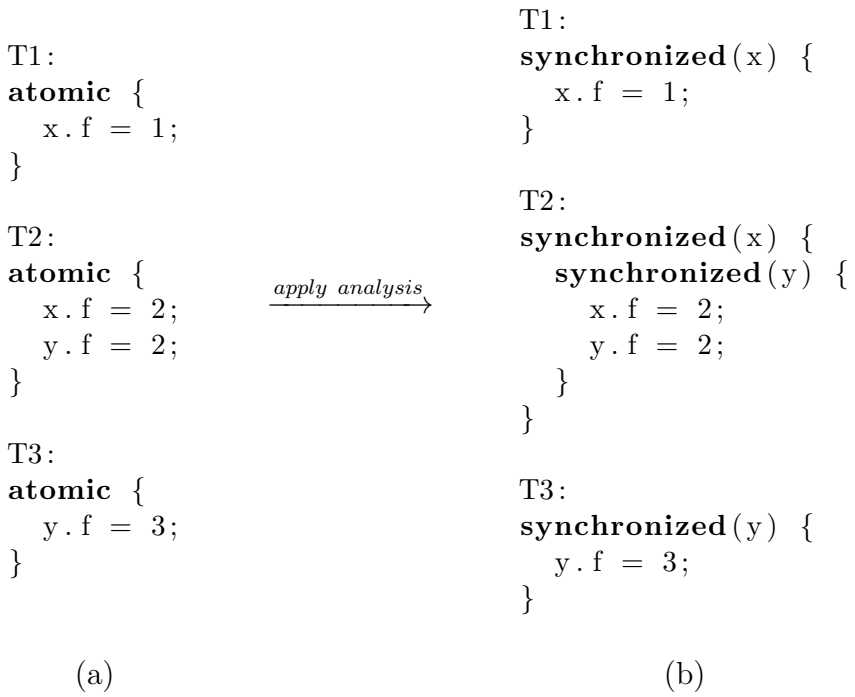


Figure 2.13: An example illustrating the general idea behind lock inference.

2.5.1 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 resulting program is serialisable and does not deadlock.

To illustrate this, consider the example program in Figure 2.13(a). It consists of two shared 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

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

Figure 2.14: Iterating through a dynamic data structure. It is not possible to know at compile-time how many objects will be accessed at run-time.

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 in Figure 2.13(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.

2.5.2 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 2.14 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.

	[MZGB06]	[HFP06]	[EFJM07]	[ZSZ+08]	[HPV07]	[CGE08]	[CCG08]
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
Libraries	Pre-locking	No	No	No	One-level deep	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	Yes (preempt)	No
Evaluation							
Large examples	Yes	No	Yes	Yes	Yes	No	Yes
Run-time results	Yes	No	No	Yes	Yes	No	Yes

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

Data representation

There are two representations inferred by existing lock inference work. One approach is to infer *abstract objects* [HFP06, HPV07]. 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 [Pea05].

The second approach is to infer *lvalues* [MZGB06, CGE08, EFJM07, CCG08]. 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:

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

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

(a)

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

(b)

Figure 2.15: Assignments (a) and aliasing (b) affect which lvalues are inferred.

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 allocation sites.

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 2.15(a). The object being updated in the second line is `me.account`. However, `you.account` is assigned to `me.account` before the update. Hence, with respect to the start of the atomic section, the object being updated is actually `you.account`.

In [CGE08, CCG08], lvalues are rewritten as they are pushed up the atomic section while [MZGB06] 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.

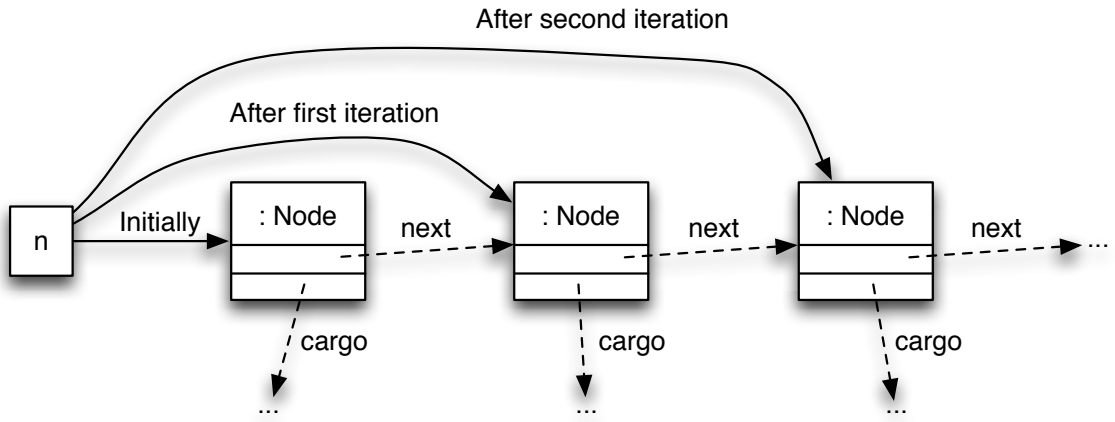


Figure 2.16: Heap-centric view of iterating through a linked list.

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 2.15(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 [MZGB06] assumes that all non-global lvalues of the same type are aliases, while [CGE08] 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 [HFP06] uses coarse locks when aliasing makes it unclear which objects are being accessed. [EFJM07] distinguishes between must- and may-aliases and uses this information to impose constraints 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.

Unbounded accesses

We revisit the linked list traversal example of Figure 2.14. 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 a possible run-time heap organisation of the linked list in Figure 2.16 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? [CCG08] caps the number of field lookups in lvalues, while in [CGE08], we infer nondeterministic finite state automata, which are equivalent to regular expressions. For example, the set above can be written as `n(.next)*`.

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 [MZGB06, HPV07, HFP06] is to ignore such thread-local accesses. With [MZGB06], it is implicit because they assign locks to data that could potentially be shared, whereas with the other two [HPV07, HFP06], 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 [HG06] is to distinguish between accesses made inside atomics and accesses made outside. This means that if some data is only accessed by one atomic, regardless of whether it is thread-local or thread-shared, there is no need to protect it. Of course, it would be need to be checked that that atomic section itself cannot be executed by concurrent threads.

Libraries

Libraries are an important component of any real-world programming language. However, their complexity and size make statically analysing them a real challenge both in terms of memory requirements and time. Libraries have a number of features that make them challenging:

- **Cyclomatic complexity:** Libraries contain long call chains as well as large set of mutually recursive methods. For example, in Oracle's JDK, we have found mutually recursive groups with over 2000 methods. These large sets of recursive methods cause scalability problems and lead to tremendous imprecision in analysis results[5].
- **Generality:** Libraries are designed to be general and handle all possible usage scenarios. For example, the `println()` method must be able to print different character sets. This requires calling into character set loading and encoding components when necessary. However, most of the time, a default character set will be used. Furthermore, static analysis has to be conservative and assume that a different character set could be loaded and must therefore include these code paths. This adds a kind of imprecision into analysis results but they will contain code paths that are not commonly executed.
- **Source code:** Libraries are usually provided in binary form. As a result, source code analyses will not be able to analyse them.

The first two points are what make analysing libraries most challenging and is why prior lock inference approaches have waved their hands. There are four main approaches that are taken when tackling libraries:

[5] Imprecision is caused by dataflow information being propagated through invalid paths (see Section 2.4.2)

- **Ignore them:** This is the common approach whereby library method calls are essentially treated as no-ops [HFP06, EFJM07, ZSZ+08, CCG08, CGE08].

- **Pre-locking:** In Autolocker [MZGB06], library method parameters that need to be protected are annotated `$locked`. The analysis then ensures that these annotated parameters are locked before the method is called.

- **Analyse up to one-level deep:** Halpert et al [HPV07] analyse library call chains upto one level deep and rely on original library synchronisation beyond that. There are many programs where this is sufficient, however code which has deep library call chains fails. Furthermore, if there is no synchronisation present in the library then their approach does not guarantee safety of library accesses at all. For instance, we ran their tool (r3043) on a concurrent version of the Hello World program (shown in Figure 2.17), having removed existing synchronisation from the library and observed that because they only analyse one level deep, they inferred empty read and write sets. Running the resulting program resulted in print buffers being corrupted causing strings to be printed out multiple times or not at all. The output of their tool when run on this example is given in Chapter A.

- **Use hand-crafted summaries:** Another approach not employed by prior work but which could be is to construct hand-crafted summaries of the effects of library methods. This is tricky because one has to ensure that all shared accesses are accounted for, which might not always be possible due to encapsulation.

As a result, all of the prior approaches are unsound because they may allow some library accesses to go unprotected, leading to atomicity violations. Given that even simple programs can involve large amounts of library code, this is a serious problem and the one that we tackle in this thesis.

2.5.3 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

```
class ConcurrentHelloWorld {
    public static void main(String[] args) {
        Thread[] threads = new Thread[8];
        for (int i=0; i<8; i++) {
            threads[i] = new Thread() {
                public void run() {
                    for (int i = 0; i < numPrints; i++) {
                        atomic {
                            System.out.println("Hello World!");
                        }
                    }
                }
            }
        }
        for (int i=0; i<8; i++) {
            threads[i].start();
        }
        for (int i=0; i<8; i++) {
            threads[i].join();
        }
    }
}
```

Figure 2.17: Concurrent hello world example to demonstrate how Halpert et al [HPV07] 's treatment of the library can lead to unsoundness.

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 locking granularity. We now look at these areas.

Isolating conflicting atomic sections

[HPV07] identify that existing lock inference work can be categorised as being either top-down or bottom-up. Top-down approaches [HPV07, ZSZ+08] 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. [HPV07] improve upon this by also considering which atomic ections could actually execute concurrently, using a refined *May-Happen-in-Parallel* analysis [NA98].

Bottom-up approaches [MZGB06, HFP06, EFJM07, CGE08, CCG08] on the other hand, begin

```
struct entry { int k; int v; struct entry *next; };

mutex table_lock;
struct entry *table[SZ] protected_by(table_lock);

void put(int k, int v) {
    int hashcode = ...;
    struct entry *e = malloc(...);
    e->k = k;
    e->v = v;
    atomic {
        e->next = table[hashcode];
        table[hashcode] = e;
    }
}
```

Figure 2.18: Example from Autolocker [MZGB06] demonstrating their `protected_by` annotation for associating locks with shared data.

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 but have the advantage that they could have more flexible locking policies.

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 approaches [HFP06, EFJM07, ZSZ+08, HPV07, CGE08, CCG08] performs this mapping automatically, with the exception of [MZGB06], 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. Figure 2.18 shows an example hashtable written in C from their paper that uses the `protected_by` annotation to associate a lock with the hashtable.

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 [HFP06], 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 [MZGB06, CGE08, CCG08, EFJM07], however, aliasing [EFJM07] and unbounded accesses [CGE08, CCG08] often mean that coarser locks are used. A possible solution is to use locks of differing 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 [CGE08, CCG08].

Finally, top-down approaches to lock inference [ZSZ+08, HPV07] 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.

Minimising the number of locks

A number of approaches also employ additional techniques to reduce the number of locks. [EFJM07] and [ZSZ+08] use 0-1 ILP and formulate lock inference as an optimisation problem. [ZSZ+08] also use heuristics, such as "all conflicting atomic sections must have one lock in common." [HPV07] also use heuristics. [HFP06] coalesce locks which are always acquired together.

2.5.4 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.

Locking policy

We have already seen that the locking policy must be two-phased to ensure serialisability. The basic version of acquiring all locks at the start of the outermost atomic section and releasing them at the end is used by the approaches of Hicks et al [HFP06], Zhang et al [ZSZ+08], Halpert et al [HPV07] and Cherem et al [CCG08]. McCloskey et al [MZGB06] and Emmi et al [EFJM07] use late locking whereas Cunningham et al [CGE08] experiment with early unlocking.

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, [MZGB06] 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. [EFJM07], who

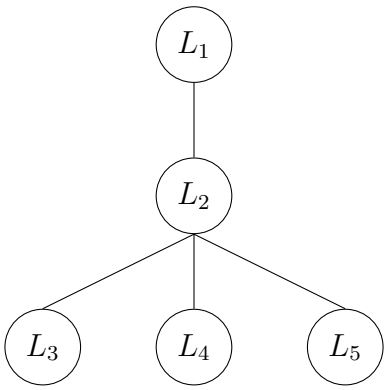


Figure 2.19: An example multi-granularity locking hierarchy whereby multiple child locks L_3, L_4 and L_5 have the same ancestors. [CCG08] ensure deadlock-freedom by ensuring that all ancestor locks are acquired but they do not give details of how they would prevent deadlock for a hierarchy like this one.

extend [MZGB06], also order lvalues but use global locks when this is not possible. Other approaches which statically order are [HFP06] and [ZSZ+08]. [HPV07] uses static (i.e. compile-time) locks when deadlock is possible.

[CGE08] and [CCG08] differ from the aforementioned approaches in that they avoid deadlock dynamically. [CGE08] 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. [CCG08] on the other hand, claim that by ensuring all ancestors in the multi-grain lock hierarchy are already locked then deadlock is avoided. However, they do not explain how deadlock is avoided between multiple child locks that have the same ancestors, as shown in Figure 2.19.

A note about lock order

It was mentioned above that locks need to be acquired in some global order to avoid deadlock. However, for approaches that infer lvalue expressions, deadlock is not the only problem that can occur by acquiring locks in the wrong sequence.

Suppose you have a path expression `x.f.g`. Dereferencing this involves first resolving `x` and then the two successive field lookups for `f` and `g` respectively. During this resolution, any field

yet to be looked up can be modified by a concurrent thread. This is fine as those fields have not been read yet. However, fields that have already been resolved may also change. This may be problematic because it means that the final object obtained will be different to what `x.f.g` now points to. Figure 2.20 shows a possible heap organisation for this example. There are two threads concurrently executing: `T1` that is resolving `x.f.g` and `T2` who is assigning to `x.f`. `T1` successfully resolves `x.f` and currently holds a reference to object `b1`. Thread `T2` then comes along and assigns `x.f` the object `b2`. `T1` performs the final field lookup for `g` to obtain a reference to object `c1`, however, this is out-of-date as `x.f.g` now points to `c2`. This means that if `T1` were to subsequently lock `c1` thinking that it was what `x.f.g` pointed to, it would have acquired the wrong lock. If locks are acquired at the start of the atomic section, this could lead to a safety violation because when it then actually accesses `c2` by re-resolving the path expression in the atomic section, it will not be holding a lock on it and so a race condition could occur.

To avoid the wrong locks being taken, it is necessary to acquire them in *prefix order* (e.g. acquire locks in the order `x`, `x.f` and `x.f.g`). This prevents a field already resolved from being modified. However, with this lock ordering constraint, it may not be possible to impose a global ordering to prevent deadlock. This tension is avoided by most prior work: [MZGB06] rejects programs for which a global ordering is not possible, [HPV07], [EFJM07], [HFP06] and [ZSZ+08] use a finite set of global locks that is entirely known at compile-time. [CGE08] on the other hand does acquire locks in prefix order. They detect deadlock at run-time and retry acquiring locks if it occurs.

2.5.5 Additional features

We finally look at additional features supported by some approaches.

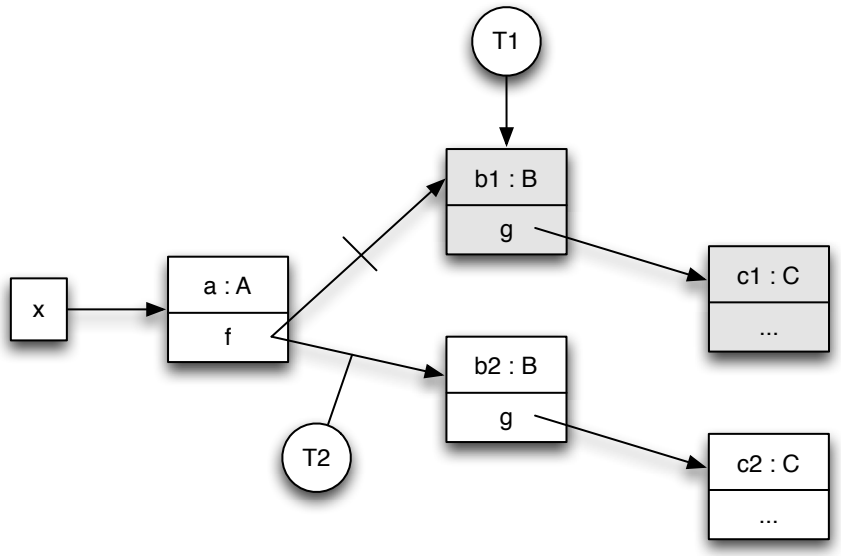


Figure 2.20: Example illustrating that the resultant object from resolving a path expression such as `x.f.g`, can be incorrect if previously resolved fields are modified by concurrent threads. Here, thread `T1` resolves `x.f` to object `b1` but thread `T2` subsequently changes it to point to `b2`. As a result, `T1` resolves `x.f.g` to `c1` whereas it is now `c2`. If `T1` subsequently locks `c1`, it would be the wrong lock for protecting the access of `x.f.g`.

Truly nested atomic sections

Almost all lock inference approaches use a flat nesting model whereby nested atomic sections are merged with their parent, creating one large atomic section. This can negatively hit concurrency. The exception to this is [HPV07] 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 [Mos06].

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 shared data to be modified. This could break atomicity. Conditional variables are supported by [MZGB06, HPV07, ZSZ+08]. [CGE08] introduce a `preempt` construct that splits the atomic

```
class ConditionVariable {
    LinkedList<Thread> waiters = new LinkedList<Thread>();

    public atomic void wait() {
        Thread t = Thead.currentThread()
        waiters.add(t);
        preempt {
            LockSupport.park();
        }
    }

    public atomic void notify() {
        if (!waiters.isEmpty()) {
            Thread t = waiters.removeFirst();
            LockSupport.unpark(t);
        }
    }

    public atomic void notifyAll() {
        while (!waiters.isEmpty()) {
            notify();
        }
    }
}
```

Figure 2.21: Implementation of a condition variable using Cunningham et al's `preempt` construct [CGE08].

section into two, which they use to implement condition variables. When a `preempt` region is encountered, all already-acquired locks are released and re-acquired once execution of the `preempt` region has completed. Figure 2.21 shows the resulting implementation of a condition variable. This approach unfortunately breaks atomicity and would therefore require further evaluation to determine how useful it would be.

2.6 Soot

We now briefly describe the Soot framework, which we use to implement our lock inference techniques. Soot [VRCG+99] is a Java optimisation framework for analysing and transforming Java bytecode. It reads in Java source or bytecode and can transform it to one of four intermediate representations. The most commonly used of which is *Jimple*, a typed 3-address code

```
int[] x = new int[12];
x[1] = 2;
```

(a)

```
0:  bipush  12
2:  newarray int
4:  astore_1
5:  aload_1
6:  iconst_1
7:  iconst_2
8:  iastore
```

(b)

```
r1 = newarray (int)[12];
r1[1] = 2;
```

(c)

Figure 2.22: (a) is an example Java snippet that creates an array an initialises the second element, (b) is the corresponding bytecode and (c) is the Jimple version. Jimple is a typed 3-address code representation used by the Soot framework.

representation. Figure 2.22 shows an example Java snippet, it's corresponding bytecode and Soot's jimple representation. As you can see, Jimple is much closer to the original source and makes writing analyses simpler in comparison to the stack-based machine of bytecode.

Soot also contains a number of useful analyses already implemented within it, such as call-graph construction, context-sensitive and context-insensitive points-to [LH03, LH08] and use-def.

2.7 Conclusion

In this chapter, we have visited a number of background areas to give the reader a solid grounding for the remaining technical portions of this thesis. In particular, we have looked at the history and semantics of atomic sections, transactional memory and program analysis. Finally, we surveyed all prior lock inference approaches.

A significant universal weak-spot is the handling of libraries. Libraries are an important component of any real-world language and if lock inference is to be a serious implementation of atomic sections, it is necessary for techniques to be able to scale to them. Furthermore, lock inference has the advantage that it can support irreversible operations such as I/O and system calls. However, as shown by the Hello World program, these irreversible operations use large portions of the library and so again lock inference needs to be able to support them.

This is not an easy task because libraries make static analysis difficult due to their cyclomatic

complexity and generality. They require the developing of special techniques, which is the main contribution of this thesis: a set of analyses that enable lock inference for general Java programs making arbitrary use of the library. In particular, ours is the first approach that is able to fully analyse library call chains and thus infer a sound set of locks for an atomic section. In addition to this, we also apply a number of novel techniques to reduce the number of locks inferred such as finding *instance-local* objects.

We begin by introducing our basic analysis for inferring object accesses.

Chapter 3

Scalable Lock Inference

Programming languages typically come with a rich set of libraries that provide common functionality, such as maintaining a hashtable or performing I/O. They are usually very large and written in a very general manner. This makes static analysis extremely difficult [RSX08], as an analysis, to be correct, must consider all possible code paths, even if a large proportion of them are executed very infrequently. This leads to very long analysis times and lots of imprecision in analysis results. An analysis may not even be able to complete due to insufficient memory. This is a significant problem for lock inference approaches because most real programs make extensive use of libraries. Although long analysis times is not that problematic, as the results would only be computed once, actually being able to analyse the library and reducing the imprecision that the library introduces *are* important problems. Furthermore, one of the major advantages of lock inference is that it allows irreversible operations such as system calls and I/O. However, from the Hello World example in Section 1.6, we have seen that these operations rely on large parts of the library. This again reiterates the need for a serious lock inference implementation to be able to handle libraries.

Because of the complexity that libraries present, prior work has either ignored them or dodged them by either annotating which locks to take or only analysing library call chains upto one level deep. The main contribution of this thesis is a set of techniques that can scale to large Java libraries. In this chapter we describe our overall approach to lock inference and our

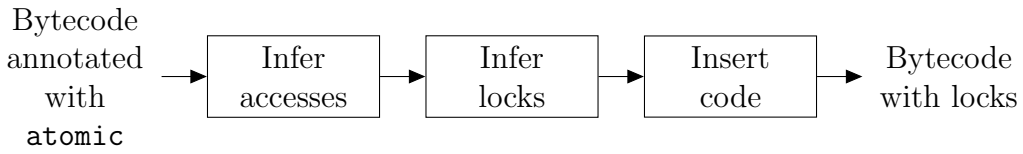


Figure 3.1: Overview of our lock inference analysis

basic analysis for inferring which objects are accessed inside atomic sections. In the next two chapters we will describe optimisations we employ to reduce the space and time requirements of our analysis as well as analyses to reduce the number of locks inferred.

3.1 General approach

Our general approach is to use the Soot framework [VRCG+99] to analyse Java classes annotated with atomic sections and replace these annotations with suitable locks. Our analysis ensures weak atomicity (global lock semantics) and consists of three stages, which are shown in Figure 3.1.

First, we perform a dataflow analysis to infer what objects are accessed in each atomic section. Nested atomics are flattened and merged with the outermost one. We compute summaries for each method, which describe the accesses performed by it and all transitively called methods. The result of the analysis is a graph at each program point p, describing objects accessed between p and the end of the atomic section.

The graph computed at the start of the atomic section describes all objects accessed in it, which we convert to locks. Where possible, we infer *instance locks*, however, for those portions of the graph that describe a statically unbounded set of accesses (e.g. due to a linked list traversal), we infer locks on the *types* of these objects. We use *multi-granularity locking* [GLP75] to support both kinds of locks simultaneously: a type lock can be acquired if none of the locks on its instances are currently acquired and vice-versa.

Finally, we instrument the program with the inferred set of locks, such that they are acquired upon entry to the atomic section and released upon exit. Acquiring all locks together at the

```
class Scheduler {
  Printer p1, p2;

  atomic boolean schedule(Job j) {
    if (p1.job == null) {
        p1.job = j;
    }
    else if (p2.job == null) {
        p2.job = j;
    }
  }
}
```

(a)

```
class Scheduler {
  Printer p1, p2;

  boolean schedule(Job j) {
    lockRead(this)
    lockWrite(p1);
    lockWrite(p2);
    if (p1.job == null) {
        p1.job = j;
    }
    else if (p2.job == null) {
        p2.job = j;
    }
    unlockWrite(p2);
    unlockWrite(p1);
    unlockRead(this)
  }
}
```

(b)

Figure 3.2: A simple example of how our analysis would transform an atomic section. Here, a scheduler has two printers. As we don't know at compile-time which printer object's job field will be written to, we have to conservatively assume both could and therefore infer write locks for both. (a) is the original version and (b) is our transformed version.

start, allows us to test for deadlock at run-time. If it occurs, we release all locks that have already been acquired and subsequently attempt to re-acquire them. As no updates have been performed this is safe.

Figure 3.2 shows an example atomic method and the lock operations that would be instrumented by our analysis. The example consists of two `Printer`s and a `Scheduler`, which allocates a given job to the next available `Printer` (each of which can only handle one job at a time). Statically, we can't be sure which conditional branch will be executed, so we must acquire a write lock on both `Printers`.

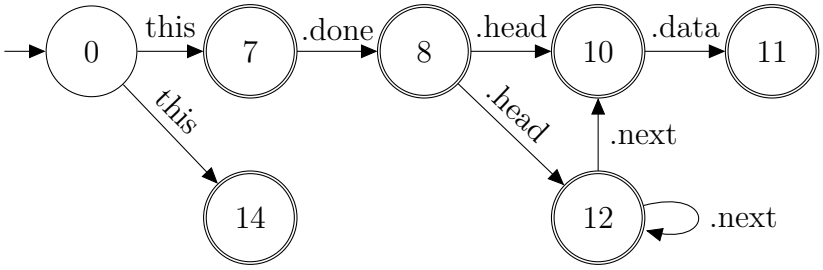
The rest of this chapter is organised as follows: In section 3.2, we present our object access inference analysis that can scale up to Java programs that use the class library and in section 3.3 we discuss how to infer locks from our analysis.

```
1 class Printer {
2   LinkedList<Job> pending, done;
3   int pendingCount, doneCount;
4
5   atomic float calcAvgWaitTime() {
6     int totalWait = 0;
7     LinkedList<Job> jobs = this.done;
8     Node<Job> n = jobs.head;
9     while (n != null) {
10      Job j = n.data;
11      totalWait += j.elapsed;
12      n = n.next;
13    }
14    return (float)totalWait/this.doneCount;
15  }
16 }
```

Figure 3.3: Printers with queues.

this
this.done
this.done.head
this.done.head.data
this.done.head.next
this.done.head.next.data
...

(a) Paths set



(b) Non-deterministic Finite Automaton (NFA)

Figure 3.4: Inferred non-deterministic finite automata from the atomic `calcAvgWaitTime` method in Figure 3.3.

3.2 Inferring Object Accesses

We infer syntactic expressions of the form `x.f`, also known as *paths* [CA04, CGE08], that evaluate at run-time to object references. Each object is protected by its own lock, so a path expression can also be used to obtain this unique lock (as in Java). However, the set of paths accessed by an atomic section may be unbounded. For example, when traversing a linked list, at compile-time we cannot know in general how may times the loop will be iterated and can only assume it may be infinite. We overcame this in previous work [CGE08] by representing sets of paths as *non-deterministic finite automata (NFA)*. NFAs are equivalent to regular expressions and give us a precise, finite representation.

In Figure 3.3, we modify the printer example of Figure 3.2 so that printers instead have a queue

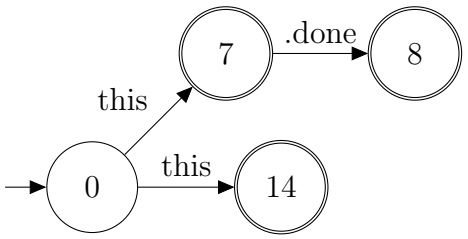
of pending jobs. The `Printer` class also has a method `calcAvgWaitTime()` that returns the average waiting time across all completed print jobs. This method is `atomic` because the `done` list and associated `doneCount` field should not be modified during the calculation. The set of paths accessed and the equivalent NFA that our analysis infers are shown in Figure 3.4.

A widely-used technique for interprocedural dataflow analysis, is the *functional* approach [SP81]. Dataflow values are translated in one step using a method m's summary function, which cumulatively describes how m transforms dataflow information. Library methods only have to be analysed once and their summaries can be stored for re-use later when analysing client programs. Summary functions are computed by composing the individual transfer functions for each of m's statements. During this computation, the dataflow information are these transfer functions. To be able to compute summaries scalably, it is essential to have a compact representation for transfer functions with fast composition and meet operations.

We formulate our analysis as an *Interprocedural Distributive Environment* (IDE) [RSX08] analysis. As described in Section 2.4.2, dataflow values in an IDE analysis are mappings of type $D \to L$ called *environments*. D is a finite set of symbols and L is a finite height meet semilattice. Transfer functions describe how statements transform environments and are called *environment transformers* (called *transformers* for short). If $Env(D, L)$ is the set of all environments for a given D and L, then transformers have type $Env(D, L) \to Env(D, L)$. Furthermore, transformers have the special property that they are *distributive*. That is, $\forall t \in Env(D, L) \to Env(D, L)\ .\ \forall e_1, e_2 \in Env(D, L)\ .\ t(e_1 \sqcap e_2) = t(e_1) \sqcap t(e_2)$.

The advantage of this framework is that a compact representation of transfer functions exists that allows fast composition and meet. Sagiv et al [SRH96] represent transformers as graphs, allowing composition to be computed by taking the transitive closure and meet by graph union or intersection. Rountev et al [RSX08] have also shown that IDE analyses with this representation can scale well when using large Java libraries.

In the next section, we begin formulating our IDE analysis by first defining our environments.



(a)

this $\mapsto \{(0,7),(0,14)\}$
.done $\mapsto \{(7,8)\}$

(b)

Figure 3.5: (a) Portion of automaton from Figure 3.4 and its environment representation (b)

3.2.1 From sets to environments

In [CGE08], we represent an automaton as a set of edges. Hence, the first step is to represent them as environments instead; that is, mappings from some finite set of symbols D to elements of a finite height meet semi-lattice L. To describe this process, we use the definition of an automaton as a five-tuple: $(Q, \Sigma, \delta, q_0, F)$, where Q is the set of states; Σ is the set of transition labels consisting of local variables, fields, classes (for static accesses) and [*] (for array accesses); and δ is the transition function. Our IDE analysis represents an automaton as a mapping from transition labels $l \in \Sigma$ to their corresponding transitions (represented as pairs of the form (q_1, q_2) where $\delta(q_1, l) = q_2$). Let $StatePairs = Q \times Q$. Thus, we choose $D = \Sigma$ and $L = \mathcal{P}(StatePairs)$. Note, L is finite because Q is finite [CGE08]. Figure 3.5 shows a portion of the automaton of Figure 3.4 and its corresponding representation as an environment.

3.2.2 Environment transformers

Environment transformers describe how program statements translate dataflow information, which we now define for our analysis.

We acquire all locks at the start of the atomic section. This allows us to test for deadlock at run-time but is challenging because it means that the object referred to by a path, such as `x`, may differ between the point where `x` is dereferenced and the point where locks are acquired (i.e. at the beginning of the atomic section), due to assignments that occur in-between (see Section 2.5.2 for more details). Our transformers translate paths accordingly to preserve the set of objects that are accessed, albeit potentially introducing new accesses due to the conservatism

$$
\begin{array}{rcl}
t_{[\texttt{x = y}]^n} & = & \lambda e.e[y \mapsto e(y) \cup e(x)][x \mapsto \emptyset] \\
t_{[x=\texttt{null}]^n} & = & \lambda e.e[x \mapsto \emptyset] \\
t_{[x=\texttt{new}]^n} & = & \lambda e.e[x \mapsto \emptyset] \\
\hline
t_{[x=y.f]^n} & = & \lambda e.e[y \mapsto e(y) \cup \{(0,n)\}] \\
 & & \quad [.f \mapsto e(.f) \cup \{(n,n') | (0,n') \in e(x)\}] \\
 & & \quad [x \mapsto \emptyset] \\
\hline
t_{[x.f=y]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}] \\
 & & \quad [y \mapsto e(y) \cup \{(0,n'') | (n',n'') \in e(.f)\}] \\
t_{[x.f=\texttt{null}]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}] \\
t_{[x.f=\texttt{new}]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}] \\
\hline
t_{[x=y[*]]^n} & = & \lambda e.e[y \mapsto e(y) \cup \{(0,n)\}] \\
 & & \quad [[*] \mapsto e([*]) \cup \{(n,n') | (0,n') \in e(x)\}] \\
 & & \quad [x \mapsto \emptyset] \\
\hline
t_{[x[*]=y]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}] \\
 & & \quad [y \mapsto e(y) \cup \{(0,n'') | (n',n'') \in e([*])\}] \\
t_{[x[*]=\texttt{null}]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}] \\
t_{[x[*]=\texttt{new}]^n} & = & \lambda e.e[x \mapsto e(x) \cup \{(0,n)\}]
\end{array}
$$

Figure 3.6: Environment transformers for object access inference

of our alias analysis.

Figure 3.6 contains our transformers, which we now describe in turn. We use Soot's three-address Jimple representation (see Section 2.6). Each CFG node is labelled with a unique identifier n. We represent a CFG node in text with the notation $[...]^n$

$[x = y]^n$ The object referenced by `x` after this assignment is actually that pointed-to by `y` before the assignment. Hence, to preserve object accesses performed lower down via paths beginning with `x`, they must be rewritten to begin with `y` instead. For example, in `atomic { x = y; x.f = 1; }`, the access of `x` in `x.f` requires locking `y` at the start of the atomic section. We achieve this by modifying the incoming environment e by replacing all automaton transitions of the form $0 \xrightarrow{x} n'$ with $0 \xrightarrow{y} n'$. This involves copying `x`'s transitions to `y`'s set: $\texttt{y} \mapsto e(y) \cup e(x)$, and deleting `x`'s transitions: $\texttt{x} \mapsto \emptyset$.

$[x = new]^n$ **and** $[x = null]^n$ In these two cases, accesses of `x` below the assignment will either be local to the atomic section (`new`) or generate a `NullPointerException` (`null`). No locks need to be acquired, so we delete paths beginning with `x` by removing all $0 \xrightarrow{x} n'$ transitions:

$\texttt{x} \mapsto \emptyset$.

$[x = y.f]^n$ The transformer for this statement performs two tasks. Firstly, it records that the object pointed-to by `y` is being accessed, by adding the transition $0 \xrightarrow{y} n$ to the incoming environment e: $y \mapsto e(y) \cup \{(0, n)\}$. Secondly, it preserves object accesses performed via paths prefixed with the variable `x` by rewriting them to start with `y.f` instead. For example, in `atomic { x = y.f; x.g = 1; }`, to protect the object access `x` in `x.g` at the start of the atomic section, we require locking `y.f`. This is achieved by replacing all transitions of the form $0 \xrightarrow{x} n'$ with the pair of transitions $0 \xrightarrow{y} n$ (already generated above) and $n \xrightarrow{.f} n'$: $.f \mapsto e(.f) \cup \{(n, n') | (0, n') \in e(x)\}$. Finally, we delete `x`'s transitions: $\texttt{x} \mapsto \emptyset$.

$[x.f = y]^n$ This statement accesses the object `x` and modifies its `f` field to point to object `y`. Our transformer records the access by adding it to `x`'s transition set in the incoming environment e: $x \mapsto e(x) \cup \{(0, n)\}$.

With previous statements, we preserve object accesses made below by simply rewriting paths beginning with the lvalue to instead be prefixed with the rvalue. This assignment is not as straightforward because it could, in addition to paths starting with `x.f`, also affect paths prefixed with `z.f` for all variables `z` that alias `x`. For example, in `atomic { x.f = y; z.f.g = 1; }`, to protect the access `z.f` in `z.f.g`, there are two possibilities. (i) `x` and `z` are aliases: the atomic section is then the same as `atomic { z.f = y; z.f.g = 1; }`, so the object referred by `z.f` after the assignment is actually `y` before the assignment, so we lock `y`. (ii) `x` and `z` are not aliases: the object `z` is not modified by the first assignment, therefore the path `z.f` is not affected so we lock `z.f` (and not `y`).

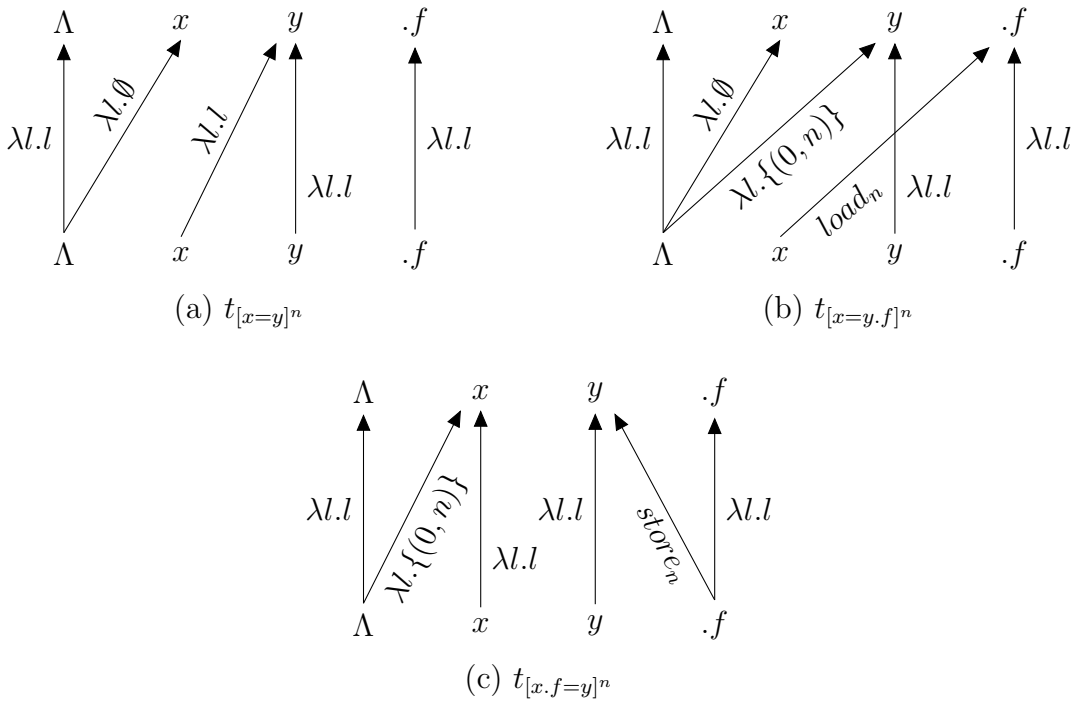
Our analysis uses type information to determine whether two paths may alias each other. In particular, the assignment `x.f = y` affects the path `z.f` if the classes that define the field `f` being accessed in both `x.f` and `z.f` (determined statically in Java) are the same. If they are, we add the path `y`, otherwise we conclude that `z.f` will definitely not be affected and do nothing. Note, even if `x` and `z` may be aliases, the original path `z.f` is not deleted in case they're not.

In general, the affected path may be of the form `v.`$\bar{f}$`.f` where $\bar{f}$ is a sequence of zero or more field lookups that could include `f`. Hence, our transformer adds a transition $0 \xrightarrow{y} n'$ for each $n'' \xrightarrow{.f} n'$ transition whereby field f is defined in the same class as `f` in `x.f`: $y \mapsto e(y) \cup \{(0, n'') | (n', n'') \in e(.f)\}$. Points-to information would reduce the number of $0 \xrightarrow{y} n''$ transitions but may complicate the composition of transformers.

$[x.f = new]^n$ **and** $[x.f = null]^n$ As type information only tells us if two paths may alias, we can never assert that they definitely must alias. Hence, we cannot assume that accesses of the form `z.f` will be local (`new`) or generate a `NullPointerException` (`null`). We can assume this for paths prefixed with `x.f` as we know `x.f` aliases itself. In this latter case, we would not acquire the lock for `x.f`. To cover both scenarios where we can and can't delete the path, the transformer performs no translation. Note that `x` is being dereferenced so we record this: $x \mapsto e(x) \cup \{(0, n)\}$.

$[x = y[*]]^n$ The transformer for this statement is similar to that for `x = y.f`. We record the access of the array object `y` in the incoming environment e: $y \mapsto e(y) \cup \{(0, n)\}$. However, when translating, we do not distinguish between different array locations representing them all using [*], which can be read as "somewhere in the array." Our transformer preserves object accesses by translating all paths that begin with `x` to instead start with `y[*]`. That is, we replace each transition $0 \xrightarrow{x} n'$ with the pair $0 \xrightarrow{y} n$ (generated above) and $n \xrightarrow{[*]} n'$: $[*] \mapsto e([*]) \cup \{(n, n') | (0, n') \in e(x)\}$. At run-time, locking `y[*]` involves locking all elements of the array `y`.

$[x[*] = y]^n$ We assume all arrays are aliased, hence this assignment could affect all paths that end in `[*]`. When translating such paths, we cannot be sure they refer to the same array location being assigned to. Even in the case of `x[*]`, although we are certain the same array is being modified, the indices may differ. Consequently, our transformer does not delete any paths (like for `x.f = y`) but adds a transition $0 \xrightarrow{y} n'$ for each transition of the form $n'' \xrightarrow{[*]} n'$: $y \mapsto e(y) \cup \{(0, n') | (n'', n') \in e([*])\}$.



(a) $t_{[x=y]^n}$ (b) $t_{[x=y.f]^n}$

(c) $t_{[x.f=y]^n}$

Figure 3.7: Pointwise representations for the key transformers in Figure 3.6.

3.2.3 Graph representation of transformers

The scalability of a summary-based analysis depends upon the representation of transfer functions and how efficiently their composition and meet can be computed. For IDE Analyses, Sagiv et al [SRH96] show that transformers can be represented as compact graphs, called *pointwise representations*, whose composition is the transitive closure and meet is essentially graph union or intersection.

Informally, these graphs describe how the exit environment e' is derived from the entry environment e. An edge $d_1 \xrightarrow{f} d_2$ in the graph means that $e'(d_2)$ is obtained from $e(d_1)$, with edge function $f : L \to L$ describing exactly how so. In the simplest case, $f = \lambda l.l$ (the identity function), so $e'(d_2) = e(d_1)$. If $e'(d_2)$ is dependent on multiple $e(d_k)$, the meet of the values (after applying the edge functions) is taken. New values (not derived from e) are introduced using the special symbol Λ.

Figure 3.7 shows the pointwise representations for $t_{[x=y]^n}$, $t_{[x=y.f]^n}$ and $t_{[x.f=y]^n}$ from Figure 3.6 (we assume here that $D = \{x, y, .f\}$). The arrows are directed from bottom to top because our analysis is backwards. Our analysis has five edge functions:

```
atomic void enqueue(Document d) {
    Job j = new Job(d);
    j.elapsed = 0;
    d.queued = true;
    // add j to queue of pending jobs
}
```

Figure 3.8: Figure 3.3 example extended with an `enqueue` method

1. $\lambda l.\{(n', n'')\}$ for introducing a new automaton transition $n' \xrightarrow{d} n''$. For example, the statement $[x = y.f]^n$ of Figure 3.7(b) accesses object y and therefore $e'(y)$ must contain the new pair $(0, n)$. This is represented by the edge $\Lambda \xrightarrow{\lambda l.\{(0,n)\}} y$.

2. $\lambda l.\emptyset$ for killing transitions. For example, in Figure 3.7(a), $e'(x) = \emptyset$ corresponds to the edge $\Lambda \xrightarrow{\lambda l.\emptyset} x$.

3. $\lambda l.l$ for copying transitions. The edges $y \xrightarrow{\lambda l.l} y$ and $x \xrightarrow{\lambda l.l} y$ in Figure 3.7(a) collectively give that $e'(y) = e(y) \cup e(x)$ (as defined in Figure 3.6).

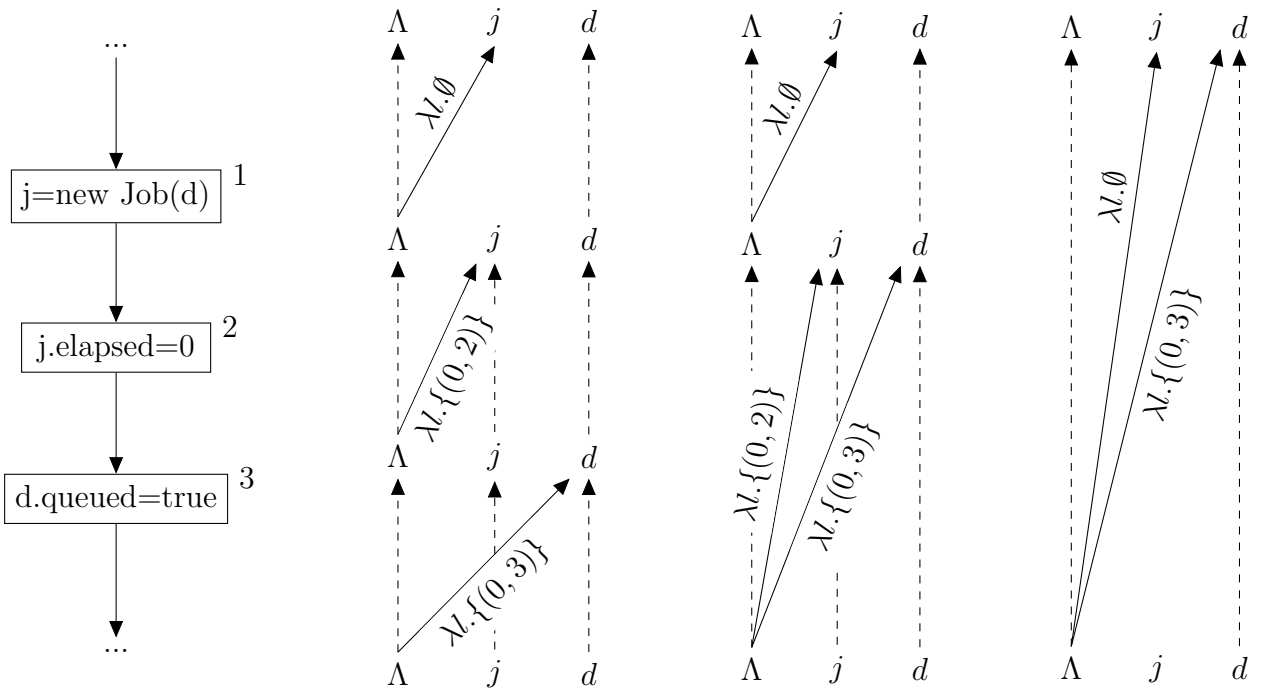
4. $load_n = \lambda l.\{(n, n') | (n'', n') \in l\}$ for preserving object accesses across statements of the form $[x = y.f]^n$ and $[x = y[*]]^n$.

5. $store_n = \lambda l.\{(0, n') | (n'', n') \in l\}$ for preserving object accesses across statements of the form $[x.f = y]^n$ and $[x[*] = y]^n$.

3.2.4 Transformer Composition

To illustrate how transformer composition works, we extend our `Printer` example from Figure 3.3 with another atomic method `enqueue` in Figure 3.8, that adds a given `Document d` to the printer's queue of pending jobs. This method needs to be atomic because concurrent threads shouldn't be allowed to modify the printer's queue or the document `d` while it is executing.

Figure 3.9(a) gives the CFG for `enqueue`. Figure 3.9(b) shows each CFG node n's transformer placed directly to n's right. Edges of the form $d \xrightarrow{\lambda l.l} d$ are called *trivial edges*. To simplify graphs, we draw them with a dashed line and omit the identity edge function.



(a) CFG (b) Initially (c) One step (d) Two steps

Figure 3.9: (a) CFG for `enqueue`. (b)-(d) show the successive results for composing transformers (performed bottom up)

Composing transformers is performed bottom up (because this is a backwards analysis), therefore we first compose $t_{[d.queued\ =\ true]^3}$ together with $t_{[j.elapsed\ =\ 0]^2}$, the result of which is shown in Figure 3.9(c). Transformer composition is computed by taking the transitive closure of edges (and composing edge functions). Figure 3.9(d) shows the result of composing the transformer computed in Figure 3.9(c) with $t_{[j\ =\ new\ Job(d)]^1}$. This resulting transformer describes the cumulative effects on dataflow information of all three statements. It has two non-trivial transitive edges $\Lambda \xrightarrow{\lambda l.(0,3)} d$ (computed by composing $\Lambda \xrightarrow{\lambda l.(0,3)} d$ with the trivial edge $d \xrightarrow{\lambda l.l} d$) and $\Lambda \xrightarrow{\lambda l.\emptyset} j$ (obtained by composing $\Lambda \xrightarrow{\lambda l.l} \Lambda$ and $\Lambda \xrightarrow{\lambda l.\emptyset} j$, where $\lambda l.\emptyset = \lambda l.\emptyset \circ \lambda l.l$).

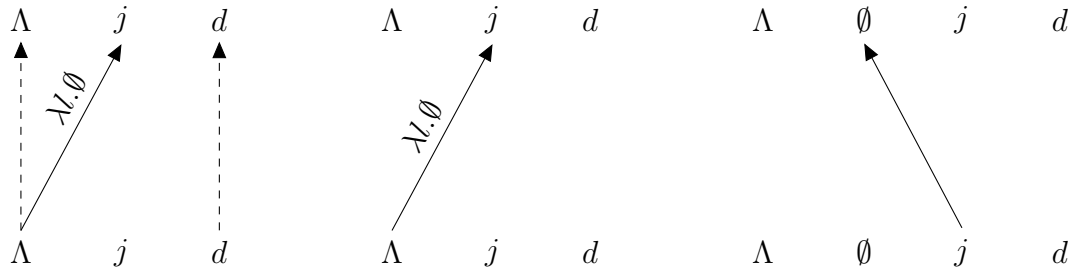
3.2.5 Sparsity

An important optimisation to reduce the size of a transformer and simultaneously the time taken to perform compositions and meets, is to keep the graphs as sparse as possible. We achieve this by not explicitly representing trivial edges (i.e. of the form $d \xrightarrow{\lambda l.l} d$). Despite not explicitly representing these edges, it is still necessary to detect that they exist when performing

transformer operations. However, it turns out that this can be costly, potentially overriding the benefits obtained from sparsely representing them. To demonstrate this, Figure 3.10(b) shows the sparse graph for $t_{[j\ =\ new\ Job(d)]^1}$ of Figure 3.9. Both d and j have no outgoing edges but while the implicit edge $d \xrightarrow{\lambda l.l} d$ exists, the same is not true for $j \xrightarrow{\lambda l.l} j$. This is because j is killed in the exit environment, as represented by the edge $\Lambda \xrightarrow{\lambda l.\emptyset} j$. Hence, to determine if a trivial edge $d_i \xrightarrow{\lambda l.l} d_i$ exists, the transitive closure now requires checking whether the edge $\Lambda \xrightarrow{\lambda l.\emptyset} d_i$ exists. This has to be done for all d_i, which will slow down transformer composition tremendously. So, although we have achieved a space reduction, we lose out in the time dimension.

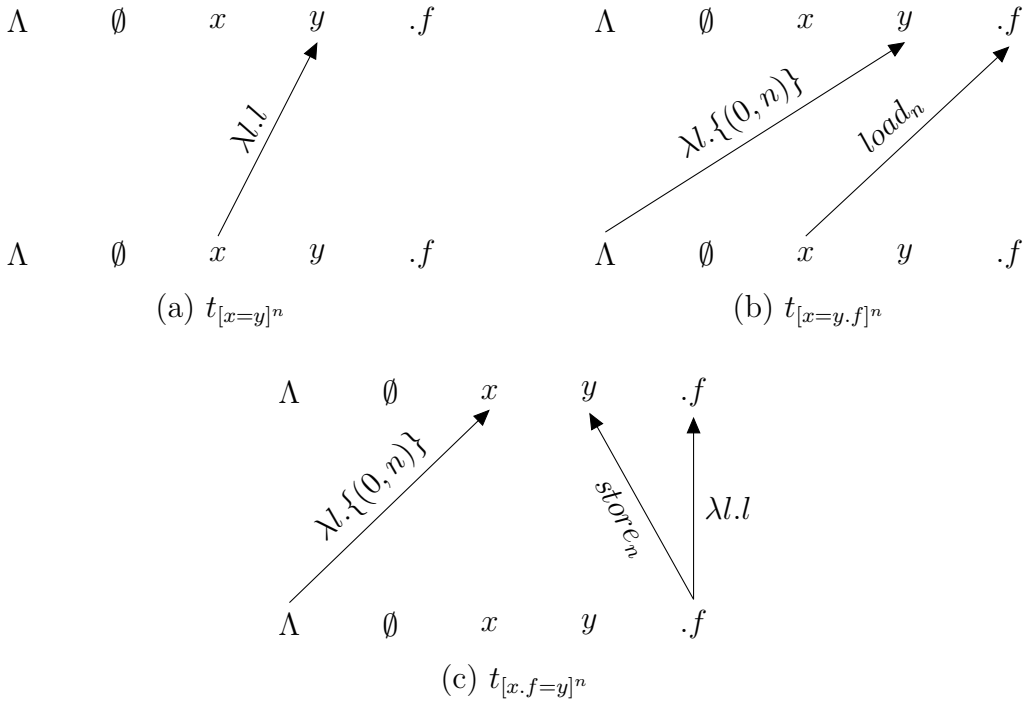
To overcome this problem, we firstly introduce a new special symbol $\emptyset$. Killing the value for symbol d_i in the exit environment is then represented with the edge $d_i \rightarrow \emptyset$. Secondly, we observe that a large majority of our transformers perform kills (i.e. replace automaton transitions), hence we implicitly encode killing within transformer edges. That is, an edge $d_1 \xrightarrow{f} d_2$ now additionally has the meaning $e'(d_1) = \emptyset$. This latter refinement removes the need for kill edges when rewriting paths (e.g. $[x = y]^n$), leading to sparser graphs. The two refinements combined yield the result that an implicit edge $d_i \rightarrow d_i$ exists iff d_i has no outgoing transitions in a transformer. Figure 3.10(c) shows the refined graph. Symbols Λ, $\emptyset$ and d have no outgoing edges and so each have trivial edges. j has an outgoing edge, therefore has no trivial edge. Figure 3.11 shows the refined sparse pointwise representations of Figure 3.7. In the case of Figure 3.7(c), as we do not kill $.f$ in the exit environment, we must add an explicit edge $.f \xrightarrow{\lambda l.l} .f$. However, statements of the form $[x = ...]^n$ are more common, hence the overall effect is that our refined transformers contain significantly less edges than the original version of Sagiv et al [SRH96].

Transformer Meet When all edges are explicitly represented, the meet of transformers is graph union. However, when edges are implicitly represented this is not the case and extra care is needed. Figure 3.12(a) gives two example transformers whose meet is to be computed. The first transformer preserves all values from the entry environment to the exit environment. The second transformer, however, copies x's value across to y before killing x's value. Hence, the



(a) All edges (b) Sparse (c) Refined sparse

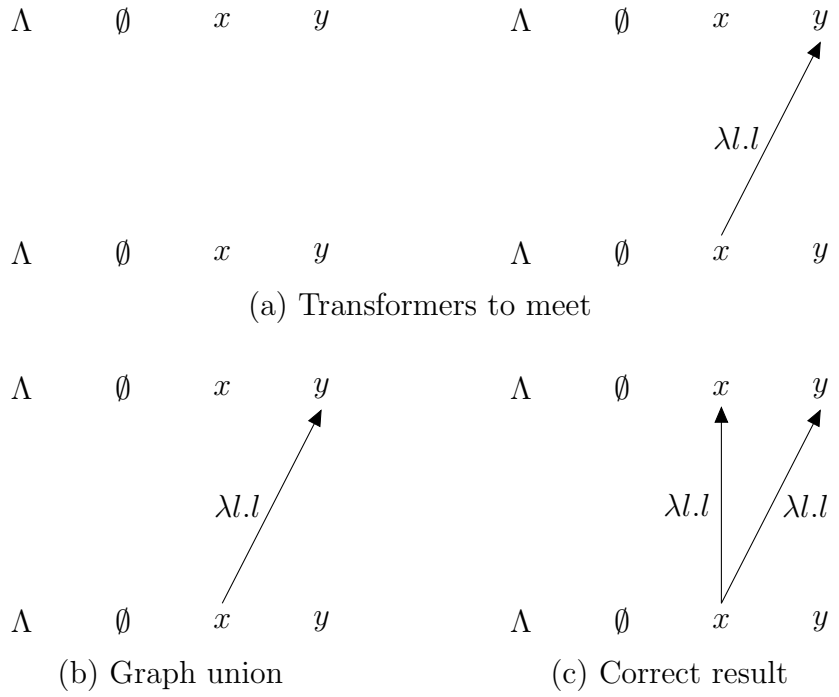
Figure 3.10: Determining whether a trivial edge exists in our sparse transformer is costly, hence we refine the representation. (a) contains the original transformer with all edges represented explicitly, (b) is the sparse version and (c) is the refined sparse version. The refinements we make are that (1) we introduce the symbol $\emptyset$ and subsequently represent killing a mapping by the edge $d_i \to \emptyset$ and (2) we implicitly encode killing in an edge. That is, the edge $d_i \to d_j$ also means that $e'(d_i) = \emptyset$. These two refinements mean that a trivial edge $d_i \to d_i$ exists if d_i has no outgoing edges.



(a) $t_{[x=y]^n}$ (b) $t_{[x=y.f]^n}$

(c) $t_{[x.f=y]^n}$

Figure 3.11: Refined pointwise representations for Figure 3.7



(a) Transformers to meet

(b) Graph union

(c) Correct result

Figure 3.12: Computing the meet when implicit edges are present

combined transformer should both preserve x's value and also copy it to y. Figure 3.12(b) shows the resulting transformer after union, which is not the desired result. This is because graph union is oblivious to the fact that x has an implicit edge in the first transformer. To resolve this, our meet operation makes a trivial edge explicit if at least one other transformer doesn't also implicitly have it. However, if none of the transformers have the trivial edge, then it isn't generated in the merged result. The correct result for this example is shown in Figure 3.12(c).

3.2.6 Computing method summaries

Code within atomic sections may invoke methods. To infer object accesses across method boundaries in a scalable way, we compute a summary for each method m that describes its object accesses as well as how it cumulatively transforms dataflow information. This allows m's effects to be inlined at call sites by composing with its summary transformer. In this section, we describe how summaries are derived from the transformers computed by our analysis and also how interprocedural propagation works.

We assume each method m has a unique entry statement N_m and exit statement X_m. Further-

more, return values are represented using the ghost variable $r, i.e. $[\texttt{return x}]^n$ is treated as $[\texttt{\$r = x}]^n$ ($[\texttt{return}]^n$ is considered a no-op). Each CFG node n in m has a local transformer t_n, which describes how n transforms environments (e.g. $t_{[x=y]^n}$). Our analysis also computes an aggregate transformer t_{n,X_m} at n that summarises the transformation on environments along all execution paths between n and X_m inclusive. It is determined by first taking the meet of all aggregate transformers computed at successor nodes: $\bigsqcap\{t_{s,X_m} \mid n \to s \in m\}$ and composing this result with t_n, i.e. $t_{n,X_m} = t_n \circ \bigsqcap\{t_{s,X_m} \mid n \to s \in m\}$. Consequently, the aggregate transformer t_{N_m,X_m} computed at the entry statement N_m, describes the effects for all execution paths through the method m. However, this transformer will contain information about local variables that is irrelevant to a calling method. Hence, we remove this method-local information to yield the summary for method m, which we refer to as T_m.

3.2.7 Interprocedural propagation

We now describe how these summaries are used at call sites for interprocedural propagation. Assume method f contains the call $[x = y.m(a_1, ..., a_k)]^n$. The local transformer for this caller node encapsulates three steps: (i) parameter passing, (ii) execution of the callee method m and (iii) storing the return value to result variable x. We conceptually expand n to a series of sub-statements comprising assignments of arguments to parameters: $[this = y]^{n_{this}}$ and $\forall i : 1, ..., k\ [p_i = a_i]^{n_{p_i}}$; the method invocation $[m(this, p_1, ..., p_k)]^{n_{invoke}}$; and the assignment of the return value $[x = \$r]^{n_{result}}$. t_n is thus the composition of the local transformers for each of these sub-statements:

$$t_n = t_{n_{result}} \circ t_{n_{invoke}} \circ t_{n_{p_k}} \circ \cdots \circ t_{n_{p_1}} \circ t_{n_{this}}$$

The transformer $t_{n_{invoke}}$ is the summary of the callee m, i.e. T_m. However, due to polymorphism, there may be several possible callees for $[m(this, p_1, ..., p_k)]^{n_{invoke}}$ and as this is a static analysis, we have to assume that any could be executed. We therefore take the meet of all such callee summaries: $t_{n_{invoke}} = \bigsqcap\{\ T_m \mid m \in targets(n)\ \}$.

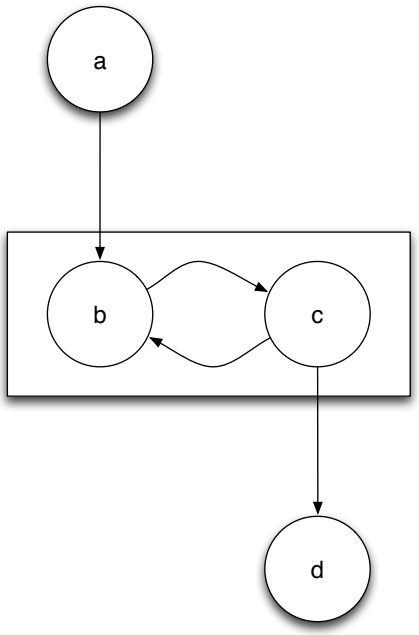
```
void a() {
    b();
}

void b() {
    if (...)
        return;
    else
        c();
}

void c() {
    if (...)
        d();
    else
        b();
}

void d() { ... }
```

(a)



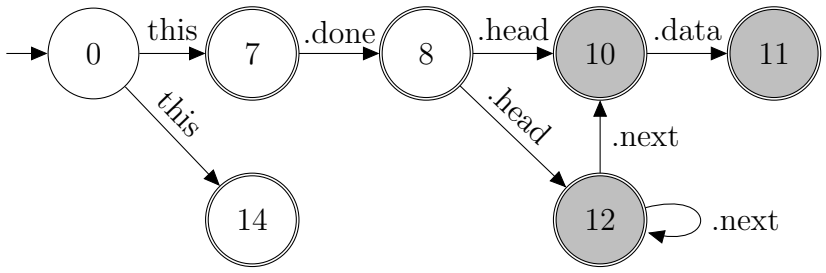
(b)

Figure 3.13: Example call-graph containing a set of mutually recursive methods.

Normally, summaries for target methods will be computed before summaries for calling methods. If the caller is involved in a recursive cycle with the callee, then this is not possible. In this case, the summary T_m is computed iteratively together with T_f. Furthermore, the invoke transformer $t_{n_{invoke}}$ is initially unknown[1] and must therefore also be calculated iteratively.

To illustrate how propagation proceeds up the call graph, consider the example program in Figure 3.13(a) and its corresponding call graph in Figure 3.13(b). As method d is the leaf of the call graph, we first compute its summary T_d. b and c are next but as they recursively call each other, their summaries T_b and T_c must be iteratively computed together, applying T_d at the call to d. Finally a's summary is computed utilising T_b. When computing multiple summaries together, such as T_b and T_c in this example, they are initially approximated as being empty and progressively refined. During this computation, every time the calls `b()` and `c()` are encountered, the current approximation of the respective summary is used and by repeatedly doing so, a fixed point is eventually reached.

[1] In general, some callees may be involved in a recursive cycle with the caller and some may not be, so $t_{n_{invoke}}$ would be partially known.



(a)

R(this)
R(this.done)
R(this.done.head)
R(Job)
R(Node)

(b)

Figure 3.14: (a) is the NFA of Figure 3.4 and (b) is the corresponding set of inferred locks.

3.3 Inferring Locks

After having computed a fixed point in our IDE analysis, we now map the inferred object accesses to locks. From the transformer computed at the start of the atomic section, we extract the NFA describing all object accesses. This NFA is constructed from all transformer edges of the form $\Lambda \xrightarrow{\lambda l.\{(0,n)\}}$. We subsequently infer a set of locks from this automaton.

Given that we assume each object is protected by its own individual lock, these path expressions can also be used to refer to this lock. However, the finite automaton may represent a statically unbounded set of paths, which is a problem because we can only infer a finite set of locks. If the set of paths is not finite, we instead lock their possible run-time types.

Our lock inference algorithm tries its best to infer per-instance locks. Hence, for the portions of the automaton that describe a finite set of paths, we infer path expressions that correspond to the instance locks and for the remaining parts, we infer the run-time types of the path expressions. We use multi-granularity locking [GLP75] to be able to support both type locks and instance locks simultaneously.

To illustrate how this process works, we revisit the automaton of Figure 3.4(b), presented again here in Figure 3.14(a) for convenience. As a pre-processing step, our analysis first identifies all automaton states that are part of, or reachable from, a cycle (these are shaded in light grey).

The lock inference algorithm starts from the start state and does a depth first traversal of the automaton. In this example, we visit states 0, 7, 8 and 10. Upon transitioning to a state, the currently accumulated path expression is extended and a new lock is added to the set for it. Hence,

so far we have inferred the read locks: `R(this)`, `R(this.done)` and `R(this.done.head)`[2]. State 10 is part of a cyclic access and therefore, we switch to inferring types. For each subsequent state we visit, we use points-to information to infer the possible run-time types of the access in the CFG node that generated the state we are currently at. For example, at state 11, we query the possible types of the access in CFG node 11 (i.e. line 11 of Figure 3.3) to obtain `Job`. Backtracking and continuing in this way, we infer the additional type read locks: `R(Job)` and `R(Node)`. Figure 3.14(b) gives the final set of locks inferred.

3.4 Avoiding deadlock

The locking policy that we instrument must not result in deadlock. Prior approaches have typically resorted to imposing a compile-time ordering on locks. However, in our case this is not possible as we do not know which locks will be taken beforehand. Another approach is to detect deadlock when it occurs at run-time by maintaining a waits-for graph. We are able to do this because we acquire all locks together at the start of the outermost atomic section and so no shared updates would have been performed if deadlock was detected. However, doing this can be expensive especially given that deadlock is rare. Instead, we take a heuristic approach by ensuring that at least one of the four necessary conditions for deadlock never occurs [sil06]. The four necessary conditions are:

1. **Mutual exclusion:** Locks can be held by only one thread at a time.
2. **No preemption:** Locks cannot be involuntarily revoked but rather have to be voluntarily released by the holding thread (i.e. by calling `unlock()`).
3. **Circular wait:** Several threads are involved in a wait cycle where they each wait on a lock held by the next thread in the cycle.
4. **Hold and wait:** Threads do not release already-acquired locks before waiting for a lock to become available.

[2]The read/write nature of each access is obtained by looking at the CFG node corresponding to the state number

```
boolean locked = false;
while (!locked) {
    Printer o1 = this;
    if (o1.ilock.tryLock()) {
        LinkedList o2 = o1.done;
        if (o2.ilock.tryLock()) {
            Node o3 = o2.head;
            if (o3.ilock.tryLock()) {
                Class o4 = Job.class;
                if (o4.tlock.tryLock()) {
                    Class o5 = Node.class;
                    if (o5.tlock.tryLock()) {
                        locked = true;
                    }
                    else {
                        o4.tlock.unlock();
                        o3.ilock.unlock();
                        o2.ilock.unlock();
                        o1.ilock.unlock();
                        waitFor(o5.tlock);
                    }
                }
                else {
                    o3.ilock.unlock();
                    o2.ilock.unlock();
                    o1.ilock.unlock();
                    waitFor(o4.tlock);
                }
            }
            else {
                o2.ilock.unlock();
                o1.ilock.unlock();
                waitFor(o3.ilock);
            }
        }
        else {
            o1.ilock.unlock();
            waitFor(o2.ilock);
        }
    }
    else {
        waitFor(o1.ilock);
    }
}

void waitFor(Lock l) {
    l.lock();
    l.unlock();
}
```

Figure 3.15: Our deadlock-free lock acquisition algorithm for the locks inferred in Figure 3.14(b).

```
void waitFor(Lock l) {
    l.lock();
    l.unlock();

    Thread currentThread = Thread.currentThread();
    currentThread.backoffInterval *= 2;
    Thread.sleep(currentThread.backoffInterval);
}
```

Figure 3.16: To minimise the chances of livelock occurring during lock acquisition, we add an exponential backoff. Each thread has a backoff interval value which is initialised to a random value between 0ms and 10ms every time a lock is successfully acquired and multiplied by two every time a lock is not available.

The first condition cannot be avoided because we need mutual exclusion when updating shared data. The second condition also cannot be broken because revocation would require the ability to rollback an atomic section which we cannot do. Eliminating the third condition would require detecting when a wait cycle has been created by maintaining a waits-for graph, which is costly. However, we can preempt the fourth condition by ensuring threads wait on a lock with empty hands, that is they release any already-acquired locks before they block. Recall that as no memory updates have been performed yet, this is safe. When the desired lock becomes available, we can re-acquire all locks from the beginning. We are essentially rolling back the locking phase when we discover that a lock is not available but delaying re-execution until it becomes so.

We now illustrate this algorithm through an example: Figure 3.15 shows our deadlock-free acquisition loop for the locks inferred in Figure 3.14(b). We have extended `java.lang.Object` with a field `ilock` that stores a reference to the object's instance lock. Additionally, we extend `java.lang.Class` with a field `tlock` that references the type lock for the type represented by each instance of `Class`. The loop proceeds by acquiring locks one at a time. If a lock l cannot be acquired, all previously acquired locks are released before waiting for l to become free. Once l becomes free, the acquisition loop restarts. To avoid blocking if a lock is not available so that we can cleanup first, we use the non-blocking `tryLock` method [Lea05]. This tries to acquire the lock and if it succeeds returns true otherwise returns false. After having released already acquired locks, we wait for `l` to become available (see the `waitFor` method), using the blocking

`lock` method. This suspends the current thread and wakes it once `l` is available. Once woken, we could hold on to `l` and re-acquire the locks we just released. However, as explained in Section 2.5.4, locks must be acquired in prefix-complete order. Hence, we immediately release `l` and restart the loop.

The alert reader will notice that there is a possibility of livelock, whereby two or more threads continuously rollback their respective locking phases because they each need a lock that the other is currently holding. We minimise the chance of this occurring by using an exponential backoff before restarting the loop. Figure 3.16 shows our modification to the `waitFor` method to achieve this. We modify `java.lang.Thread` to have a `backoffInterval` field that records how many milliseconds the thread should wait before attempting to re-acquire locks. This backoff interval is initialised to a random value (e.g. between 0ms and 10ms) every time a lock is successfully acquired and multiplied by two every time a lock acquisition fails.

This loop acquisition algorithm breaks the necessary "hold and wait" condition for deadlock. However, the overhead that arises from blocking until `l` becomes available and for the backoff can be costly. In Section 5.3 we implement an optimisation that instead polls `l` a few times first in case it becomes available very soon after `tryLock` returned false.

3.5 Evaluation

We now present experimental results for our basic lock inference approach. Our experimental machine is called *ax3*. It has 32 8-core 2.67GHz Intel Xeon E7-8837 CPUs totalling 256 cores, 3TB RAM and runs SUSE Linux Enterprise Server 11. For running our analysis, we use Oracle's 64-bit JVM version 1.6.0_26-b03 with a minimum/maximum heap size of `60GB`. The library we analyse against is GNU Classpath 0.97.2.

For running the resulting instrumented programs, we use a commodity machine called *liatris*. It consists of an 8-core 3.4GHz Intel Core i7-2600 CPU, 8GB RAM and runs Ubuntu 11.04. We use a modified version of the production build of Jikes RVM version 3.1.1+svn (r16068M) for executing the programs.

We provide a simple implementation of multi-granularity locks that internally use Java's `synchronized` mechanism. Furthermore, we use `ThreadLocal` to store thread-local information such as locks currently acquired.

We begin by giving results for Hello World and then demonstrate that our approach can scale to a full library by analysing GNU Classpath. Finally, we apply our approach to real-world workloads in the form of a set of benchmarks. We chose the benchmarks used by Halpert et al [HPV07, Hal08] to enable a comparison.

3.5.1 Hello World

In Section 1.6, we showed that although the "Hello World!" program may appear to be a simple one-liner, it requires analysing 1150 methods from the library. Previous work does not fully analyse libraries, hence it is not clear whether existing work can handle this program. Using our own previous work [CGE08], we found it intractable. However, with the techniques described in this chapter we have been able to perform a full analysis of all 1150 library methods!

The running times (in seconds) for the path and lock inference analyses are given in Figure 3.17(a). The *Total* column gives the time it took to run the whole analysis including Soot-related costs, such as building the call graph and performing the points-to analysis. The number of instance read, instance write, type read and type write locks inferred are given in Figure 3.17(b). Memory usage peaks at 50.1GB and averages 25.6GB.

Some interesting features can be extracted from this table. Firstly, although a large number of locks are inferred, 80% of them are read locks. Furthermore, 60% are fine-grained instance locks. However, the large number of type write locks is alarming. In Chapter 5 we will look at several optimisations to reduce the number of locks and dramatically reduce the execution time.

To evaluate the execution time of Hello World instrumented with our locks, we create a benchmark in which 8 threads execute the Hello World atomic section 1000 times each. The resulting times are shown in Figure 3.17(c). The *manual* column gives the time for executing with the

(a) **Analysis (secs)**			(b) **Locks**				(c) **Run-time (secs)**		
			Instance		*Type*				
Paths	Locks	Total	Read	Write	Read	Write	Manual	Global	Ours
4452.49	1.43	4757	215	54	148	34	0.29	0.31	3.81

Figure 3.17: Analysis results for the "Hello World!" program first introduced in Section 1.6

(a) **Library Info**		(b) **Analysis (secs)**		(c) **Locks**			
				Instance		*Type*	
Package	Methods	Paths	Locks	R	W	R	W
gnu	16882	250.270	12.272	16536	6235	7510	1310
java	13815	4021.647	106.31	30065	9940	30007	5354
javax	14088	8.804	2.209	7640	3307	0	0
org	2794	0.800	0.322	1275	401	0	0
sun	28	0.034	0.13	11	4	0	0
Total	47607	4281.556	121.243	55527	19887	37517	6655

Figure 3.18: Analysis results for GNU Classpath 0.97.2p10

original locking policy of the library. The *global* column gives times for when using a single global lock across all atomic sections. The table shows that our approach is 13x slower than the original locking and 12.3x slower than using a single global lock.

Although our analysis time is high and uses a large amount of memory, the key thing to note at this stage is that this is the first time that a lock inference techniques has successfully been able to analyse this many library methods and produce locks for a program involving I/O and system calls. In subsequent chapters, we will describe techniques to bring both the analysis footprint and execution times significantly down.

3.5.2 GNU Classpath

To evaluate scalability, we analyse the entire GNU Classpath 0.97.2p10 library as packaged in Jikes RVM. It consists of 47607 non-private methods and totals about 122 KLOC. We analyse each of these non-private methods in turn[3], treating it as an atomic method. We re-use summaries if they have been computed already (during the current analysis run).

The analysis takes 1 hour and 20 minutes. Memory usage peaks at 49.7GB and averages 29.3GB.

[3]Private methods are analysed implicitly with non-private callers.

Figure 3.18 gives a per-package breakdown of: (a) number of methods; (b) path inference and lock inference analysis times in seconds and (c) gives the number of each type of lock inferred.

The method which takes the longest to analyse is `java.io.InputStreamReader`'s constructor, namely 3435 seconds. Upon inspection, we find that this pulls in a similar part of the library as Hello World. However, once this set of methods has been analysed, the summaries for methods called by most other methods has already been computed and so do not have to be recomputed. The remaining methods are analysed in a fraction of the time (average of 18ms).

From the locks inferred (Figure 3.18(c)), it can be observed that 78% are read locks. This is crucial, as it means that most accesses can proceed in parallel. Furthermore, although nearly 40% of all locks are types, 85% of them are read locks. This again is promising, because it implies that coarse grained locking would not necessarily cripple concurrency (although in the case of Hello World above, we see that the type write locks do).

3.5.3 Benchmarks

We apply our techniques to a selection of benchmark programs and compare our results with the closest known existing work of Halpert et al [HPV07].[4] The purpose of using benchmarks is to emulate real-world workloads to get a feel for how well our approach may work in practice. We choose all benchmarks from their paper that do not use wait/notify (our implementation does not currently support this) and provide analysis and run-time statistics for each. We treat all synchronized blocks and methods as if they are atomics and translate them using our algorithm. For a fair comparison when comparing against manual, global and Halpert et al, we replace synchronized blocks with calls to `lock()` and `unlock()` on our locks instead (we maintain the original locking policy).

[4] We do not use their published work [HPV07] but their later improved version [Hal08] that they kindly made available to us. This infers sets of fine-grained locks per atomic whereas in their published version they inferred at most one lock per atomic.

Comparison with Halpert et al.

An important difference between our approaches is that we analyse library methods in full whereas Halpert et al only consider accesses upto one level deep in library call chains and rely on original library synchronisation beyond that. Their approach can thus be unsound (see Section 2.5.2). In Figure 3.19(a)(i), we list the number of client and library methods called by atomic sections for each benchmark. This table shows that programs do indeed make extensive use of libraries with library methods comprising over 95% of the total methods called in some cases. Figure 3.19(a)(ii) compares analysis times (both columns for Halpert and us respectively include Soot-related costs). We give a breakdown for the running time of each component in our analysis in Figure 3.19(b).

Figure 3.20 gives a comparison of locks inferred. Figure 3.20(i) are the locks inferred by Halpert et al, Figure 3.20(ii) the locks we infer. Halpert et al distinguish between two types of lock: (i) *static* locks are known at compile-time and (ii) *dynamic* locks are the same as instance locks. Static locks are not equivalent to our type locks because acquiring a type lock implicitly locks all instances. That is, there is no relationship between static and dynamic locks in their approach. Furthermore, all locks are write locks.

Figure 3.19(a)(iii) gives execution times. We are noticeably slower for all benchmarks due to the larger number of locks being acquired. However, the breakdown in Figure 3.19(a)(i) shows that in some cases 95% of call-graph methods are not analysed by Halpert, whereas we have analysed the call-graph in its entirety and are the first lock inference approach to do so! As a result, our analysis infers many more accesses than Halpert's does. In Chapter 5 we present several optimisations to significantly reduce both the number of locks inferred and the resulting execution time to very near that of the original benchmark version.

Program	Threads	Atomics		(i) Methods		(ii) Analysis (secs)		(iii) Run-time (secs)			
		Total	Reachable	Client	Library	Halpert	Ours	Manual	Global	Halpert	Ours
sync	8	2	2	0	0	22	331	69.14	71.22	72.69	74.61
pcmab	50	2	2	2	15	22	315	2.28	3.15	2.28	12.47
bank	8	8	6	6	7	22	327	20.89	19.50	35.69	30.88
traffic	2	24	19	4	63	24	340	2.56	4.22	2.65	91.42
mtrt	2	6	4	67	1324	29	5741	0.80	0.82	0.78	0.95
hsqldb	20	240	158	2107	2955	48104	?	3.25	3.12	3.25	?

(a)

Program	Paths (secs)	Locks (secs)	Total (secs)	Avg. Memory (MB)	Peak Memory (MB)
sync	0.122	0.14	331	8022	15360
pcmab	0.246	0.092	315	7903	15367
bank	0.247	0.129	327	8013	15799
traffic	1.695	0.2	340	8118	15659
mtrt	5378.79	8.596	5741	27293	51118
hsqldb	?	?	?	?	?

(b)

Figure 3.19: Analysis and run-time results comparison for a selection of benchmarks from Halpert et al [HPV07, Hal08]. (a) is an overview of analysis and execution times and (b) gives a breakdown of the time taken for each part of our lock inference analysis. The locks column in (b) gives the time taken to convert NFAs to locks.

Program	(i) Halpert		(ii) Ours			
	Static	*Dynamic*	*Instance*		*Type*	
			R	W	R	W
sync	0	2	1	2	0	0
pcmab	0	3	1	5	0	0
bank	0	3	0	12	0	0
traffic	0	19	33	67	0	0
mtrt	1	0	905	268	726	130
hsqldb	2	11	?	?	?	?

Figure 3.20: Locks inferred by our analysis for our benchmarks alongside those inferred by Halpert et al.

3.6 Conclusion

In this chapter we presented our basic analysis for inferring which objects are accessed inside an atomic section. The key feature of this analysis is that it is able to fully analyse the entire Java library. This is significant because lock inference prides itself on being able to handle I/O and system calls. However, these irreversible operations rely on large parts of the library (as was seen from the "Hello World!" program introduced in Section 1.6). Libraries make static analysis hard and that is why prior work has resorted to either ignoring them, annotating library parameters or only analysing library call chains upto one-level deep. All of these approaches may lead to shared accesses remaining unprotected and subsequently race conditions. Ours is the first sound technique to fully analyse library methods and infer locks that cover all possible accesses that could occur. However, our basic approach is still not able to handle very large code bases such as hsqldb. Furthermore, we infer a very large number of locks for the programs that we can currently analyse, which cripples run-time performance. In the next two chapters we tackle these shortcomings by firstly, employing a number of optimisations to reduce analysis space and time requirements (Chapter 4) and secondly performing analyses to reduce the set of locks inferred (Chapter 5).

Chapter 4

Analysis optimisations

In Chapter 3, we introduced our basic analysis for inferring which objects are accessed in atomic sections and mapping these accesses to locks. However, although this initial analysis is able to analyse the entire GNU Classpath library, it still is not able to scale to very large code bases such as hsqldb. To overcome this limitation, we employ a number of optimisations to reduce the space and time requirements of our analysis. Before describing our optimisations, we first remind the reader about how we propagate dataflow information and compute method summaries. After introducing each technique, we use the Hello World program as a test bed to evaluate their individual and combined effectiveness.

For each atomic section a, we first compute summaries for all methods invoked. This is done by performing a bottom-up traversal of a's callgraph to ensure that a method's summary is only calculated once summaries for all called methods are known. The summaries $T_{m_1}, \ldots, T_{m_k}$ for mutually dependent methods $\{m_1, ..., m_k\}$ must be computed together. We therefore organise and traverse the callgraph to support both of these requirements by (i) identifying the strongly connected components (SCC), (ii) creating a directed acyclic graph with edges representing dependencies between components (SCC-DAG) and (iii) performing a post-order traversal of this SCC-DAG.

Each component c corresponds to a group of mutually dependent methods whose summaries need to be computed together. We calculate for each CFG node n in each method m in c,

its aggregate transformer t_{n,X_m}. The summary T_m is then obtained from t_{N_m,X_m} by removing method-local information. Aggregate transformers are computed using a worklist algorithm with two worklists: *intra* and *inter*. Intra consists of nodes whose aggregate transformer needs to be recomputed because the aggregate transformer of at least one intraprocedural successor has changed. Inter contains call nodes n whose invoke transformer $t_{n_{invoke}}$ needs to be updated because the summary of at least one callee has changed. If $t_{n_{invoke}}$ changes as a result, n's aggregate transformer also needs to be recomputed. Per CFG node information is only needed during summary computation after which only the method's summary is kept. Initially, intra contains the exit statement X_m of each method m in c. Either list is processed exclusively until it becomes empty.

Our memory requirements consist of storing for each CFG node n, a local transformer t_n and an aggregate transformer t_{n,X_m}. Changes made to t_{n,X_m} must be propagated to the entry statement N_m through all intermediate nodes. The updated summary T_m is then spread to all call sites in the current component.

There can be many CFG nodes and a large number of transformer edges leading to the vast memory usage and slow analysis times that were observed in Chapter 3. We employ the following techniques to reduce both memory and propagation. This will set the notation and also help understand why the memory usage and analysis times are so high.

4.1 Summarise CFGs

One approach is to reduce the number of CFG nodes. We adopt the technique in [RSX08] that summarises the effects of all execution paths between a pair of CFG nodes n_1 and n_2 by combining transformers for statements along these paths. This summary is called a *jump transformer* $t_{n_1 \to n_2}$ and allows dataflow information to be propagated from n_2 to n_1 (backwards analysis) in one step by composing with it. Calculating the meet of all successors' aggregate transformers then becomes $\bigsqcap\{t_{n \to s} \circ t_{s,X_m} \mid n \to s \in m\}$. That is, compose each successor's aggregate transformer with the corresponding jump transformer and then take the meet.

```
1 class Printer {
2   ...
3   atomic void incElapsed() {
4     incElapsedAux(pending.head);
5   }
6   atomic void incElapsedAux(Node<Job> n) {
7     if (n != null) {
8       Job j = n.data;
9       j.incElapsed();
10      Node<Job> next = n.next;
11      incElapsedAux(next);
12    }
13  }
14 }
15
16 class Job {
17   ...
18   atomic void incElapsed() {
19     int oldElapsed = this.elapsed;
20     this.elapsed = oldElapsed + 1;
21   }
22 }
```

Figure 4.1: `Printer` example extended with `incElapsed`() that increments the elapsed time of each pending job.

In the best case, we can reduce the CFG for a method m to just the two nodes N_m and X_m, whereby the jump transformer $t_{N_m \to X_m}$ summarises the entire method. However, the effects of *recursive* method calls are only partially known. The reduced CFG for m will contain three types of nodes: N_m, X_m and recursive calls rc_i. Jump transformers are computed using a simple dataflow analysis that propagates the identity transformer from X_m and each rc_i up the CFG until either N_m or rc_j is reached. We refer to X_m and rc_i as *jump targets*.

We illustrate this technique by extending our `Job` and `Printer` classes both with a method `incElapsed` that increments the elapsed time for a single job and all jobs in the pending queue respectively. In the latter case, the method walks through the linked list `pending` using the helper method `incElapsedAux` as shown in Figure 4.1.

The CFG for `incElapsedAux` is shown in Figure 4.2(a) and the reduced version with jump transformers on edges in Figure 4.2(b). Note that the call `incElapsedAux(next)` remains because it is recursive while `j.incElapsed()` is inline because it is not. The analysis initialises the jump targets X_m and 11 with the maps $X_m \mapsto id$ and $11 \mapsto id$ respectively and then

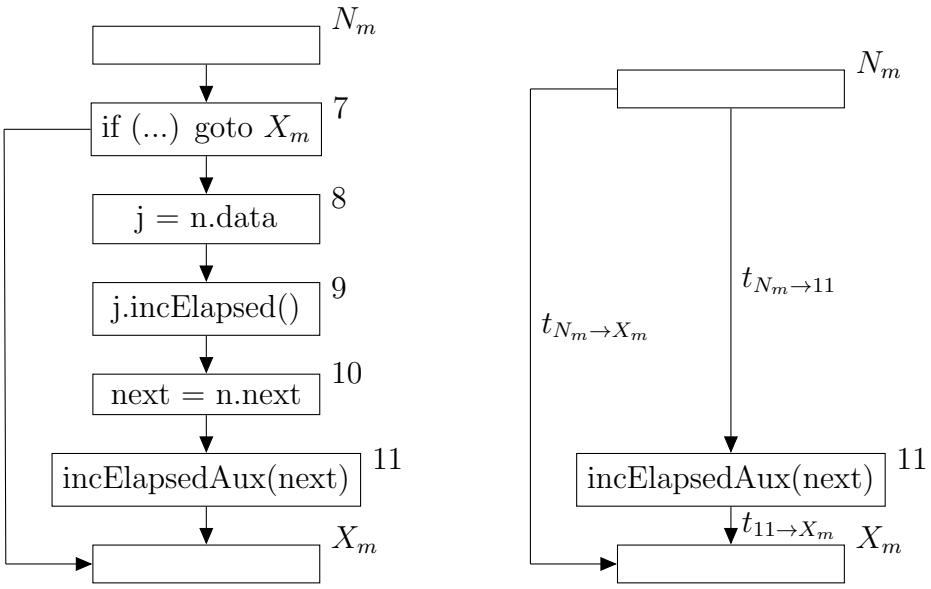


Figure 4.2: (a) CFG for `incElapsedAux` and (b) is the reduced version with jump transformers on edges that summarise the effects of all execution paths between the source and destination node. Three types of nodes remain in reduced CFGs: N_m, X_m and recursive calls.

uses a simple worklist algorithm. The result computed for each CFG node n is a map with an entry $j \mapsto t_{n \to j}$ for each jump targets j reachable from n (such that another jump target is not encountered beforehand). If n is not a jump target itself, then its transfer function is: $\lambda map.map[j \mapsto t_n \circ map(j) \mid j \in map]$. Furthermore, the meet of two maps map_1 and map_2 is computed by doing a pointwise meet: $(map_1 \sqcap map_2)(j) = map_1(j) \sqcap map_2(j)$. Once the analysis reaches a fixed point, the results at N_m and the jump targets are used to construct the reduced CFG.

4.2 Delta transformers

While evaluating analysis performance, we observed that after an initial period of propagation, transformers only grow. That is, each time a transformer $(t_n; t_{n,X_m}; T_m)$ is updated, it contains at least the edges it did previously and possibly more. This can lead to redundant work when performing transformer composition and meet. Transformer composition is distributive, hence if two edges (i.e. one from each transformer) have already been composed before, composing them again will not give a different result. Similarly, in the case of meet, unioning edges that

have already been unioned gives nothing new.

In this section, we show how transformer composition and meet can be sped up by propagating only new transformer edges. As less dataflow information information is propagated, this also reduces the amount of memory required for temporary objects. We now describe how this can be achieved.

Previously, we only stored an aggregate transformer t_{n,X_m} for each node n, corresponding to the dataflow information exiting n. We now explicitly differentiate between dataflow information flowing into and out of a CFG node using in_n and out_n respectively.[1] Assume also that our implementation stores both of these values for each CFG node n.

Suppose in_n^1 and in_n^2 are successively computed values of in_n after propagation has reached the point where transformers only grow.[2] As in_n^2 contains at least the edges in in_n^1, we can express it as:

$$in_n^2 \quad = \quad in_n^1 \sqcap (in_n^2 - in_n^1) \tag{4.1}$$

Here, $in_n^2 - in_n^1$ corresponds to transformer difference and produces a transformer containing the edges in in_n^2 that are not in in_n^1. We call this resulting transformer a *delta transformer* and denote it using $\triangle$. Hence, substituting $\triangle in_n^2 = in_n^2 - in_n^1$ into Equation 4.1 gives us:

$$in_n^2 \quad = \quad in_n^1 \sqcap \triangle in_n^2 \tag{4.2}$$

We now show how transformer composition and meet can be sped up using delta transformers. Please note that in our implementation, deltas are not computed by explicitly taking the difference but instead are constructed during the meet operation. To simplify the presentation we assume difference.

[1] Note that as per Section 2.4.1, in_n corresponds to entry information in a forwards analysis and exit information in a backwards analysis

[2] We detect this by comparing with the transformer from the previous iteration.

Composition Without delta transformers, each time in_n changes, we compose it with t_n to give the new value for out_n:

$$out_n = t_n \circ in_n \tag{4.3}$$

For out_n^1 and out_n^2, this means the following two operations are performed:

$$out_n^1 = t_n \circ in_n^1 \tag{4.4}$$

$$out_n^2 = t_n \circ in_n^2 \tag{4.5}$$

Thus, each time out_n is to be updated, a full transitive closure is performed using all the edges in t_n and in_n. However, we already know that $in_n^2 \sqsupseteq in_n^1$, so by using Equation 4.2, the second update (Equation 4.5) can be rewritten as:

$$out_n^2 = t_n \circ (in_n^1 \sqcap \triangle in_n^2) \tag{4.6}$$

Transformer composition is distributive[3], so the equation becomes:

$$out_n^2 = (t_n \circ in_n^1) \sqcap (t_n \circ \triangle in_n^2) \tag{4.7}$$

Finally, using Equation 4.3 we can make one further simplification to give:

$$out_n^2 = out_n^1 \sqcap (t_n \circ \triangle in_n^2) \tag{4.8}$$

In other words, each time in_n changes, we only need to take the composition with the new edges (i.e. the delta transformer) and union the result with the previous value of out_n. This can significantly reduce redundant work, which is important especially when transformers get

[3]Transformer composition is performed by taking the transitive closure of edges in one transformer with edges in the second transformer. As there is no dependence on other edges during each pair-wise edge composition, it follows that transformer composition is distributive.

very large.

Meet The meet operation can be optimised in a similar fashion, whereby only the meet of successors' new *out* edges is taken and added to the previous value of in_n. This again reduces the amount of redundant work and speeds up the analysis.

Suppose n has two successors s_1 and s_2. Let in_n^1 be the current value of in_n. Computing in_n^2 is then:

$$in_n^2 = out_{s_1}^2 \sqcap out_{s_2}^2 \tag{4.9}$$

Using Equation 4.8, we can rewrite this to:

$$in_n^2 = (out_{s_1}^1 \sqcap (t_{s_1} \circ \triangle in_{s_1}^2)) \sqcap (out_{s_2}^1 \sqcap (t_{s_2} \circ \triangle in_{s_2}^2)) \tag{4.10}$$

The meet operation is commutative, therefore we can rearrange as follows:

$$in_n^2 = (out_{s_1}^1 \sqcap out_{s_2}^1) \sqcap (t_{s_1} \circ \triangle in_{s_1}^2) \sqcap (t_{s_2} \circ \triangle in_{s_2}^2) \tag{4.11}$$

Using a variant of Equation 4.9, we can simplify:

$$in_n^2 = in_n^1 \sqcap (t_{s_1} \circ \triangle in_{s_1}^2) \sqcap (t_{s_2} \circ \triangle in_{s_2}^2) \tag{4.12}$$

Hence, we now only need to take the meet of the edges computed when each successor updated *out* and union it with the previous value of in_n. However, this is still not optimal as $t_{s_1} \circ \triangle in_{s_1}^2$ may contain edges that are already in $out_{s_1}^1$ and were therefore involved in the last meet. We instead obtain only the new edges as follows:

$$\triangle out_{s_1}^2 = out_{s_1}^2 - out_{s_1}^1 \tag{4.13}$$

$$\triangle out_{s_2}^2 = out_{s_2}^2 - out_{s_2}^1 \tag{4.14}$$

The final equation for updating in_n is:

$$in_n^2 \quad = \quad in_n^1 \sqcap (\triangle out_{s_1}^2 \sqcap \triangle out_{s_2}^2) \tag{4.15}$$

Intuitively, this means take the meet of each successors' new *out* edges and union it with the previous value of in_n.

Invoke Transformers We have described how delta transformers can be used to speed up the computation of in_n and out_n. Recall that for a collection of methods $m_1, \ldots, m_k$ that are recursively dependent on each other (or equivalently, are in the same strongly connected component), their corresponding summaries $T_{m_1}, \ldots, T_{m_k}$ have to be computed together (iteratively). Consequently, because the invoke transformer $t_{n_{invoke}}$ is the meet of all target summaries, if at least one target is in the same component it also has to be computed iteratively. In this section we describe how delta transformers can be used to speed up the computation of $t_{n_{invoke}}$ as well as the updating of out_n to take into account new edges in $t_{n_{invoke}}$.

Without delta transformers, $t_{n_{invoke}}$ is computed as follows (each time a callee summary changes):

$$t_{n_{invoke}} \quad = \quad \bigsqcap \{\ T_m \mid m \in targets(n)\ \} \tag{4.16}$$

Like with in_n, we can speed up this meet operation by only taking the meet of new edges and then unioning with the previous value of $t_{n_{invoke}}$. If $t_{n_{invoke}}^2$ is the current value of $t_{n_{invoke}}$, computing $t_{n_{invoke}}^3$ is then:

$$t_{n_{invoke}}^3 \quad = \quad t_{n_{invoke}}^2 \sqcap \left(\bigsqcap \{\ \triangle T_m^3 \mid m \in targets(n)\ \} \right) \tag{4.17}$$

It is possible that out_n has now changed, so the next step is to compute out_n^3. Previously, we showed that to do this, we compose t_n with $\triangle in_n$ and union with the previous value of out_n. However, recall that this relies on transformer composition being distributive. That is, if two edges have already been composed, composing them again doesn't give a different result.

$t^3_{n_{invoke}}$ may now contain new edges that have not been previously composed with edges in in_n so it would not be sound to only compute $t^3_n \circ \triangle in^2_n$. (Note, as in_n has not changed, its current value is still in^2_n). Nevertheless, we would still like to perform the minimal amount of work possible.

We know that $t^3_{n_{invoke}} \sqsupseteq t^2_{n_{invoke}}$, so we can express it as:

$$t^3_{n_{invoke}} = t^2_{n_{invoke}} \sqcap \triangle t^3_{n_{invoke}} \tag{4.18}$$

After applying parameter-to-argument renaming, we have that:

$$t^3_n = t^2_n \sqcap \triangle t^3_n \tag{4.19}$$

The equation for updating out^3_n is:

$$out^3_n = t^3_n \circ in^2_n \tag{4.20}$$

Substituting Equation 4.19 into Equation 4.20, gives us:

$$out^3_n = (t^2_n \sqcap \triangle t^3_n) \circ in^2_n \tag{4.21}$$

Distributivity of transformer composition allows us to expand this out to become:

$$out^3_n = (t^2_n \circ in^2_n) \sqcap (\triangle t^3_n \circ in^2_n) \tag{4.22}$$

After a final simplification, we get the following equation:

$$out^3_n = out^2_n \sqcap (\triangle t^3_n \circ in^2_n) \tag{4.23}$$

Hence, when $t_{n_{invoke}}$ changes, we only have to perform transformer composition with the new

in_n changes

$$
\begin{aligned}
in_n^k &= in_n^{k-1} \sqcap \left(\bigsqcap\{\triangle out_s^k \mid s \in succs(n)\}\right) \\
\triangle in_n^k &= in_n^k - in_n^{k-1} \\
out_n^k &= out_n^{k-1} \sqcap (t_n \circ \triangle in_n^k) \\
\triangle out_n^k &= out_n^k - out_n^{k-1}
\end{aligned}
$$

$t_{n_{invoke}}$ changes

$$
\begin{aligned}
\triangle t_{n_{invoke}}^k &= \bigsqcap\{\triangle T_m^k \mid m \in targets(n)\} \\
t_n^k &= t_n^{k-1} \sqcap (t_{n_{result}} \circ \triangle t_{n_{invoke}}^k \circ t_{n_{p_k}} \circ \cdots \circ t_{n_{p_1}} \circ t_{n_{this}}) \\
\triangle t_n^k &= t_n^k - t_n^{k-1} \\
out_n^k &= out_n^{k-1} \sqcap (\triangle t_n^k \circ in_n^{k-1}) \\
\triangle out_n^k &= out_n^k - out_n^{k-1}
\end{aligned}
$$

Figure 4.3: How delta transformers are used to update in_n, out_n and t_n when either in_n or $t_{n_{invoke}}$ change.

edges $\triangle t_{n_{invoke}}$ (after performing parameter-to-argument renaming) and union the result with the previous value of out_n. Intuitively, this is sound because edges in the previous value of $t_{n_{invoke}}$ (i.e. $t_{n_{invoke}}^2$) will already have been composed with edges in the current value of in_n (i.e. in_n^2) at some point previously and so composing them again would not give a different result. Figure 4.3 gives a summary of how delta transformers are used to iteratively update in_n, out_n and t_n.

Preserving Soundness A delta transformer corresponds to the difference between the current value of a transformer and its previous value. For example, $\triangle out_n^2$ contains the edges in out_n^2 that are not in out_n^1. Furthermore, we have shown that only deltas need to be propagated to predecessor CFG nodes (intraprocedural) or call statements (interprocedural). However, it is possible that out_n or T_m are updated multiple times before their corresponding deltas are propagated. An example is if the *intra* worklist is ordered such that preference is given to successor nodes (in order to reduce the amount of propagation). If a branch or loop exists, a node may be picked off the worklist several times before its predecessors are. Consequently, the delta transformer will contain only a subset of the edges that need to be propagated, leading to an unsound analysis result.

We deal with this problem by propagating $\triangle out_n$ and $\triangle T_m$ immediately to predecessors every

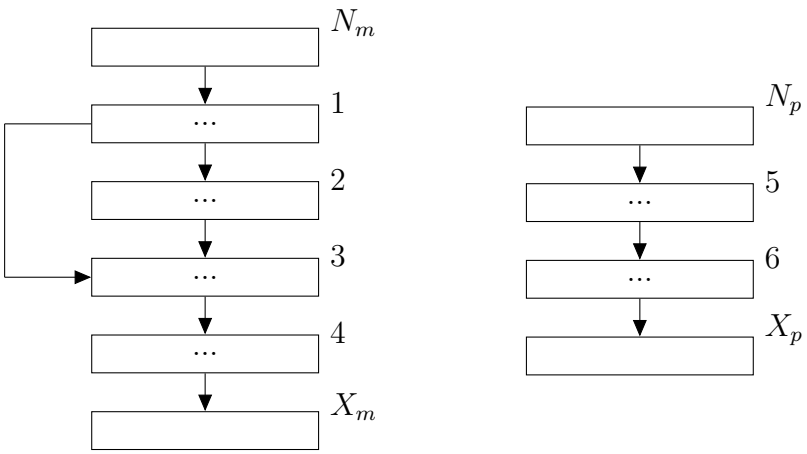
time they are computed. This requires storing for each CFG node n: $\triangle in_n$, which is updated by successors as and when they compute a non-empty value for $\triangle out$. Similarly, for call statements, we store $\triangle t_{n_{invoke}}$, which is updated by callees with their $\triangle T_m$. When a CFG node is picked off intra, it uses the stored value of $\triangle in_n$ (rather than computing the meet) after which it resets $\triangle in_n$ to the empty delta transformer. $\triangle t_{n_{invoke}}$ is handled identically when a call node is picked off inter.

4.3 Parallel propagation

Another technique we employ to speed up the analysis is to perform propagation in parallel when possible. Our *intra* worklist contains all CFG nodes that may have to be updated because at least one intraprocedural successor's out_n has changed. There is a dataflow dependency that exists between CFG nodes in the same method because data is passed from one to the next. As a result, it would be difficult to spread this propagation across multiple threads. However, we can exploit the independence of CFG nodes between different methods to construct a set of per-method worklists and process these lists in parallel. Figure 4.4 shows two example CFGs. Although, it would be difficult to parallelise propagation within method m or q, they are independent of each other and so their respective intraprocedural propagations can be performed by different threads.

Similarly, our inter worklist contains caller nodes that need to be updated because the summary of at least one callee has changed. This involves taking the meet of all callee summaries and then performing parameter-to-argument renaming. There is no dependence between different caller nodes in the list, so we process them all in parallel.

Although we parallelise both intraprocedural and interprocedural propagation, our overall propagation algorithm is *intraprocedurally eager* [KK08, KSK09]. This means that we first perform as much intraprocedural propagation as possible and only when there is no more left to do, we perform one round of interprocedural propagation. We then perform intraprocedural propagation again and this cycle continues until a fixed point is reached. The motivation behind this



(a) (b)

Figure 4.4: Although dataflow dependencies exists between CFG nodes within a method, disinct methods are independent from each other and so their respective propagations can be performed by distinct threads. (a) and (b) are two example CFGs for arbitrary methods m and p respectively.

is that interprocedural propagation is expensive and so it should be done as little as possible.

4.4 Efficient data structures

The scalability of a summary-based analysis will depend upon how efficiently its transformers can be represented and how fast the composition and meet operation can be performed. Representing transformers as graphs is a great first step as is evident from our results in Section 3.5. However, this is not enough: The choice of data structures used internally to represent these graphs can drastically impact both memory and speed.

Initially, we used `HashSet`s and `HashMap`s from the standard Java Collections API but soon found them to be less than ideal for two reasons:

- **Temporary objects:** during the analysis, a large number of temporary objects are constructed. This causes huge memory spikes and frequent garbage collections. An instance of Object in 64-bit Java occupies 8 bytes, before additional fields in subclasses are considered!

- **Lots of indirection:** We found `HashSet` contained a lot of indirection that negatively impacted both performance and memory usage. `HashSet` is implemented internally using a `HashMap`, whose hash chains are implemented as linked lists.

Representing transformer edges and their corresponding edge functions as objects also added to the number of temporary objects and indirection.

High-performance implementations typically use primitives to represent state [Lea05] and manipulate it very quickly using bit-wise operations. We represent transformer edges as 64-bit longs and implement edge composition as a bit-wise operation. However, using primitives with the Java Collections classes leads to boxing/unboxing in/out of their corresponding wrapper classes (e.g. `Long` for `long`), which again is not ideal. We therefore use the Trove library,[4] which provides primitive implementations of many Java collections such as `HashSet`s and `HashMap`s. Our transformers are then maps from integers (representing symbols) to sets of longs (representing sets of transformer edges).

Figure 4.5(a) shows our 64-bit encoding for transformer edges. The number of bits allocated for each field is shown in brackets. We now describe each field:

- **Access:** one bit is used to record whether the edge represents an object access or not.
- **Read/Write:** one bit is used to record whether this is a read or a write (set bit means write).
- **Source and destination states:** these are the source and destination automaton states respectively recorded in the edge function. The start state has the special value of all 1s.
- **Destination symbol:** the symbol this edge maps to in the exit environment e' (e.g. d_j in $d_i \rightarrow d_j$).

The fields of these fields for $load_n$, $store_n$, identity and kill edges are shown in Figure 4.5(b)-(e). These values have been specially chosen to make edge composition possible using bit-wise

[4] http://trove4j.sourceforge.net

Access (1)	Read/Write (1)	Source state (21)	Dest. state (21)	Dest. symbol (20)

(a) General edge format

0	0	n	0	d_j

(b) $load_n$ edge

0	0	*start state*	0	d_j

(c) $store_n$ edge

0	0	0	0	d_j

(d) identity edge

1	1	$2^{21}-1$	$2^{21}-1$	$2^{20}-1$

(e) kill edge (i.e. all 1s)

Figure 4.5: Our 64-bit encoding for transformer edges. (a) is the general format (number of bits for each field is shown in brackets), (b)-(e) shows the value of the fields for $load_n$, $store_n$, identity and kill edges (see Section 3.2.3 for their definitions).

operations, as shown in Figure 4.6. `composeEdges` computes $e2 \circ e1$ Moreover, `DEST_SYM_MASK` and `SRC_STATE_MASK` are bit patterns that obtain the destination symbol and source state fields respectively.

4.5 Worklist ordering

A final trivial optimisation we perform, which can make a big difference, is to order the intra worklist so that nodes lower down are given preference over those higher up. This makes intuitive sense because dataflow values propagate up the CFG (backwards analysis), and by giving preference to nodes lower down it means that dataflow values only propagate upwards once they currently cannot change any further. To illustrate this, consider the example in Figure 4.7. The details of the exact CFG nodes is irrelevant except that there is an if statement at node 1. Backward propagation starts by initialising the worklist to the method exit statement X_m. Figure 4.7(b) is the sequence of worklists that result from popping nodes off in the

```
public final long DEST_SYM_MASK = 0x00000000000FFFFFL;
public final long SRC_STATE_MASK = 0x3FFFFE0000000000L;

// post: computes e2 o e1
public long composeEdges(long e1, long e2) {
    if (isIdEdge(e2)) {
        return (e1 & ~DEST_SYM_MASK) | e2;
    }
    else if (isKillEdge(e2)) {
        return e2;
    }
    else if (isAccessEdge(e1)) {
        return (e1 & ~(SRC_STATE_MASK | DEST_SYM_MASK)) | e2;
    }
    else {
        return e2;
    }
}

public boolean isKillEdge(long e) {
    return e == -1;
}

public boolean isIdEdge(long e) {
    return (e >> 20) == 0;
}

public boolean isAccessEdge(long e) {
    return (e >>> 63) == 1;
}
```

Figure 4.6: With the 64-bit transformer edge encoding of Figure 4.5, edge composition can be performed by bit-wise operations.

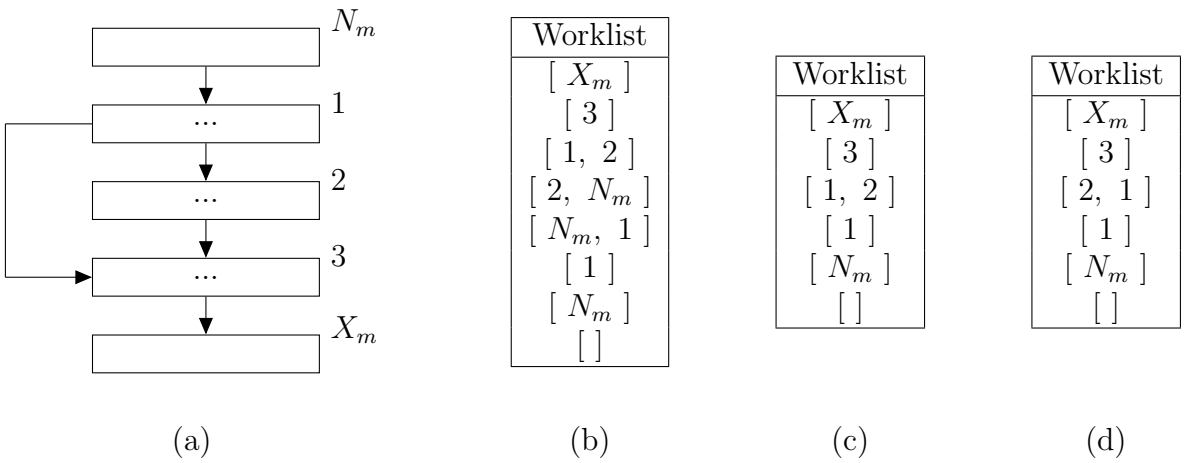


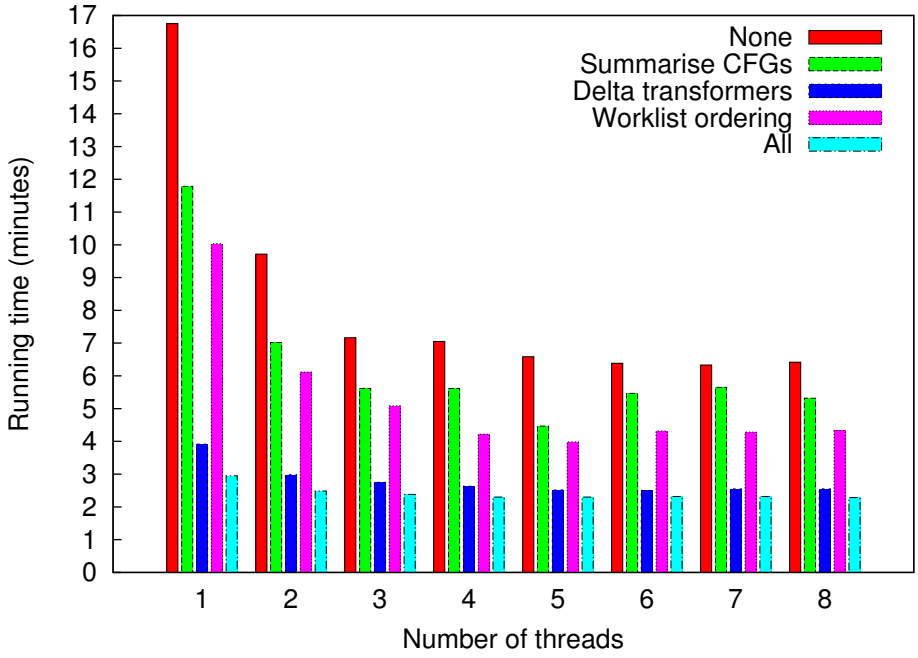
Figure 4.7: This example shows that by ordering the intra worklist such that CFG nodes lower down are given preference to those higher up, propagation can be reduced. (a) is an example CFG, (b) is the sequence of worklists that result from popping nodes off in the order they were inserted, and (c) is the sequence of worklists that result from popping off successor nodes before predecessor nodes. Worklist ordering is implemented by keeping the list sorted from highest to lowest, as shown in (d).

order they are inserted whereas in (c) successor nodes are popped off before predecessor nodes regardless of the order in which they are inserted. The important thing to note is that in the former case, node 1 is processed twice: once after node 3 and then after node 2. However, in Figure 4.7(c) it is only processed once because, when the worklist is [1, 2], 2 is given preference over it.

This ordering is achieved by partially ordering CFG nodes within a method such that higher nodes have lower numbers than lower nodes. The worklist is then kept sorted from highest to lowest. This is shown in Figure 4.7(d).

4.6 Evaluation

In this section we evaluate the impact of our analysis optimisations on memory usage and running time for when each optimisation is individually enabled. We use the Hello World atomic section to compare the effects of CFG summarisation, delta transformers, parallel propagation and worklist ordering. We also evaluate the impact of our optimisations on scalability. In particular, from Section 3.5 we saw that our basic analysis is unable to analyse the very large



(a)

Optimisation	Average MB	Peak MB
None	4923.92	8183.18
Summarise CFGs	2094.68	3470.65
Worklist ordering	4804.73	8037.14
Delta transformers	3848.98	6538.27
All	1741.39	3122.84

(b)

Figure 4.8: Effect of each individual optimisation on analysis time (a) and memory usage (b) for Hello World.

code base of hsqldb. We show that with these optimisations enabled, we are indeed now able to analyse it. Furthermore, the space and time requirements of the remaining benchmarks are dramatically reduced.

Our evaluation machine is the same as that used in Section 3.5. All analysis configurations use the efficient data structures detailed in Section 4.4. We take the average of 10 runs for each configuration.

4.6.1 Optimisation comparison

Figure 4.8(a) shows a comparison of execution times and Figure 4.8(b) a comparison of average and peak memory usage. We run each of CFG summarisation, delta transformers and worklist ordering with a varying number of threads (shown in the x-axis). The *all* configuration consists of all three optimisations enabled. When evaluating memory usage, we only use one thread.

The results give a number of interesting insights: Summarising CFGs gives the biggest reduction in memory usage. This is because the number of CFG nodes is significantly reduced and thus so is the amount of analysis state. Secondly, deltas give the best running time performance without through. In fact, it even outperforms the use of multiple threads. This is not surprising, because firstly it performs very little redundant work and secondly, as the analysis progresses, the amount of dataflow information propagated reduces thus leading to less work over time. Memory usage is also lower because the number of temporary objects are reduced.

Parallel propagation only gives gains in speed for up to three threads. We think the reason for this is because we process our two worklists one-at-a-time (see Section 4.3). Consequently, threads that have become free while processing the current list cannot proceed with the other list until the remaining threads have completed. This is essentially like a barrier operation. Some methods or caller nodes may require more propagation than others and so this creates a bottleneck.

We were surprised that ordering the worklist so that a node is given preference over its predecessors outperformed the analysis time of summarising CFGs. This might indicate that

unnecessary propagation occurs quite often if worklists are not ordered appropriately.

4.6.2 Scalability

The main contribution of this thesis is a lock inference approach that is able to scale to programs making use of libraries. In Chapter 3 we introduced our basic analysis for inferring which objects are accessed inside each atomic section and mapping these to a suitable set of locks. This basic analysis allowed us to analyse the Hello World atomic section, something which prior work has not been able to do. We were also able to analyse the GNU Classpath library entirely. However, while this been a great improvement on existing work, we saw in Section 3.5 that our basic analysis was still not able to scale to the very large code base of hsqldb. In this section, we show that with all analysis optimisations enabled we are indeed now able to analyse it. Furthermore, the running time and memory usage of the remaining benchmarks are drastically reduced.

Figure 4.9 shows the new run times for our analysis on the benchmarks *sync*, *pcmab*, *bank*, *traffic* and *mtrt* when all optimisations are enabled. Furthermore, the table also contains the results for hsqldb. Note that being able to analyse hsqldb is a big achievement, given that its call graph contains nearly 3000 library methods that are ignored by prior lock inference approaches. This is the first time that this benchmark has been analysed in its entirety. Figure 4.10 shows the number of locks inferred for hsqldb and its execution time with these locks instrumented. Our analysis was able to handle this program after enabling all our analysis optimisations and with a heap size of `70GB`. Memory usage peaked at 64.4GB and averaged 32.4GB. During the ∼7 hours taken to complete the analysis, only 153 seconds (i.e. 2.5 minutes) were spent doing garbage collection. The long analysis time is due to long call chains, large call graph components and consequently vast numbers of transformer edges that are propagated. Unsurprisingly, after the first few atomics had been analysed, the remainder were quicker because a large number of methods were common across atomics.

It is clear that the number of locks inferred for hsqldb is alarmingly high in comparison to Halpert et al. However, there are a couple of reasons for this: (1) we analyse more object accesses, as we analyse the 3000 library methods that Halpert et al do not and (2) we currently

Program	Paths (secs)	Locks (secs)	Total (secs)	Avg. Memory (MB)	Peak Memory (MB)
sync	0.053	0.0090	127	947	1781
pcmab	0.194	0.018	127	1438	2656
bank	0.151	0.019	127	1397	2868
traffic	0.433	0.059	130	946	1732
mtrt	33.901	1.902	169	1390	2618
hsqldb	21936.024	1345.859	23886	33159	65904

Figure 4.9: Shows the running time and memory usage our approach uses with all optimisations enabled for the benchmarks from Section 3.5. Both time and memory usage have dramatically been reduced. Most impressive is that we are now able to analyse hsqldb.

Program	Halpert		Ours			
	Static	*Dynamic*	*Instance*		*Type*	
			R	W	R	W
hsqldb	2	11	32508	24956	26429	10943

Figure 4.10: Locks inferred by our analysis for hsqldb alongside those inferred by Halpert et al.

assume that all object accesses are shared whereas Halpert remove locks that are thread-local. It is because of this large number of locks that the resulting instrumented program takes 500 seconds to complete. This is 160 times slower than all of a global lock, halpert's approach and the original locking policy of the benchmark.

In the next chapter, we look at several optimisations to significantly reduce this set of locks and achieve a slowdown of a mere *3.5x* for hsqldb compared to its original benchmark version.

4.7 Conclusion

In Chapter 3 we present the first lock inference analysis that is able to fully analyse library methods. We showcased its scalability by analysing the entire GNU Classpath codebase. However, our basic analysis still could not handle the very large code base of hsqldb. Although analysis times were quite slow, not being able to analyse a program is more serious. This motivated us to look for ways to improve the efficiency of our analysis both in terms of memory usage and execution time. In this chapter we present several optimisations that allow us to do this: CFG summarisation, delta transformers, worklist ordering and parallel propagation. We describe each one and then evaluate their effectiveness at improving efficiency. We also demon-

strate that with these optimisations, we are now able to analyse hsqldb. Our results show that our optimisations dramatically improve the analysis performance and memory usage.

Despite our analysis being much more efficient, the number of locks we infer is extremely high. This negatively affects the performance of the instrumented programs because there is tremendous overhead due to locking operations. In the next chapter, we shall investigate a number of techniques for improving this.

Chapter 5

Minimising locking overhead

In this thesis, we are presenting a lock inference approach for Java that is able to analyse programs that make use of the standard library. This enables us to support concurrent atomic sections that perform I/O and system calls, as they rely on large parts of the library. Due to their complexity, prior work has shied away from dealing with libraries and as a result some shared accesses may remain unprotected.

The first stage of our lock inference technique is to infer what objects are accessed in each atomic section. This set of accesses are then mapped to a suitable set of locks that protect them. Finally, the program is instrumented with these locks. In Chapter 3 and Chapter 4 we have presented our analysis for inferring object accesses, together with optimisations that are required for the analysis to scale to very large code bases. However, although we are able to achieve such scalability, the resulting performance from the instrumented programs is tremendously bad. For example, in Section 4.6, HSQLDB runs over 160x slower when instrumented with our inferred locks than the original hand-crafted version. This slowdown is due to three things:

- **Too many locks:** We assume that all object accesses are shared, however, actually most are not [CGS+99]. Furthermore, in certain cases, some shared objects also do not need to be locked.

- **Inefficient lock implementation:** Our lock implementation is built using `synchronized` and we also do not represent lock state efficiently.

- **Excessive blocking for deadlock avoidance:** When a lock l is not available, we release all already-acquired locks and then block waiting for l to be released. This breaks the necessary "hold and wait" condition and thus ensures that deadlock does not occur. However, it incurs significant overhead due to context-switches.

We now address each of these in turn. We then evaluate their effectiveness is reducing execution overhead.

5.1 Reducing the number of locks acquired

As mentioned above, our lock inference analysis assumes that all object accesses need to be locked. However, a large number of these do not need to be. We statically identify several classes of such objects: thread-local, instance-local, class-local, method-local, dominated and read-only. We also dynamically elide locks when there is only a single thread executing an atomic section. All our analyses are completely automatic and do not require any programmer-annotations. We now describe each of these, starting with lock elision.

5.1.1 Lock Elision for Single-Threaded Execution

We have found that during the initialisation of an application, a lot of objects are accessed from within atomic sections even though there is typically only a single-thread executing. Given that we perform instrumentation at compile-time, this means that locks are acquired when entering an atomic section, even though there is no contention. Such a scenario can impose significant yet unnecessary overheads on the resulting execution. Thus, we optimise our instrumentation so that locks are treated as no-ops when there is only one thread executing. Note, these object accesses are not thread-local but just that they are only being accessed by a single thread at

present. We additionally remove thread-local locks in a separate compile-time analysis, details of which are given in Section 5.1.2.

We detect whether only a single thread is executing by incrementing and decrementing a counter just before returning from `Thread.start()` and `Thread.join()` respectively. If this counter is 0 at the start of an atomic section, we elide the locks. Otherwise, we acquire them as normal.

A race could occur if a thread T1 is executing inside an atomic section with locks elided whilst another thread T2 is spawned and subsequently enters the same or another conflicting atomic section. However, this could only happen if we allowed threads to be spawned from within atomic sections (because T2 can only be spawned by T1), which we do not.

5.1.2 Thread-local objects

An object only needs to be locked if it may be accessed by multiple concurrent threads. We employ Soot's builtin thread-local analysis to detect objects that are not and do not generate locks for them. This analysis uses allocation sites to approximate run-time objects. Furthermore, as path expressions may resolve to any number of objects at run-time, we only classify a path as thread-local if all abstract objects it may point to are classified as thread-local. For details of the analysis please refer to [Lho].

5.1.3 Instance-local objects

Another class of objects we avoid locking are those that are implicitly protected and so do not need to be explicitly locked. These are objects that are completely encapsulated within an object because they exist solely to implement its functionality. Examples include the underlying `Node` objects used in Java's `LinkedList` implementation. The list dominates all accesses to the nodes, so locking the list implicitly locks the nodes. We call these *instance-local objects*, as no references to them exist outside the list. Such objects are common in Java's collections library but we have also found them common in other classes. Note, these objects may still be

```
class LinkedList {
    Node head;
    Node tail;

    public atomic void add(Object o) {
        Node n = new Node(o);
        if (head == null) {
            head = tail = n;
        }
        else {
            tail.next = n;
            tail = n;
        }
    }
}
```

```
class Node {
    Node next;
    Object cargo;
}
```

Figure 5.1: Example `LinkedList` and `Node` class definitions with an `add` method.

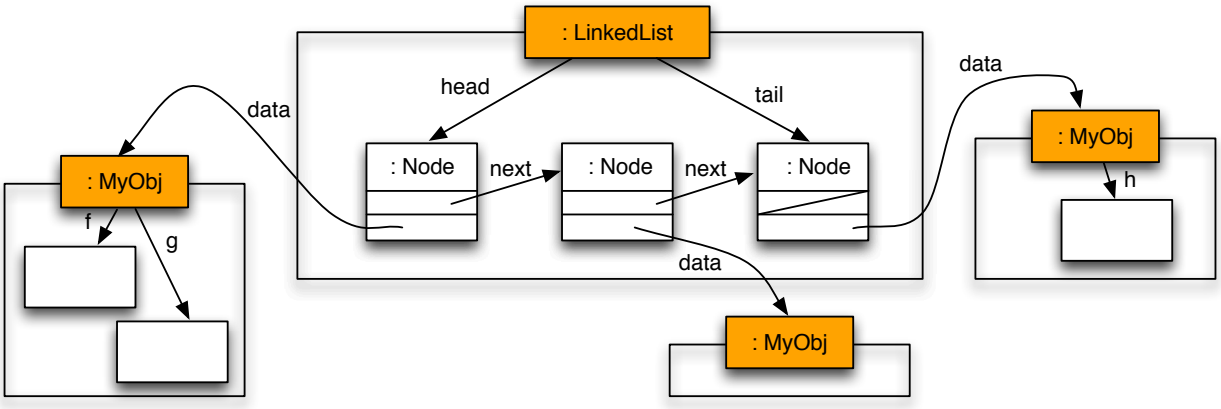


Figure 5.2: Diagram showing a possible run-time heap organisation of the list and associated objects. The list instance forms an ownership domain whereby it owns and dominates the `Node` objects within it.

thread-shared, as if the list is accessed by multiple threads then so will the underlying nodes. However, taking a lock on the list implicitly protects the nodes thus preventing the need to lock them.

Figure 5.1 gives a class definition for `LinkedList` with its `add` method. Whenever an object is added to the list, a `Node` is created to hold it and is then appended to the tail. Figure 5.2 shows a possible run-time heap-organisation of the list. Figure 5.3 gives the locks that our naive lock inference analysis inserts: A write lock on `this` is taken to protect the write of the `head` and `tail` fields. Furthermore, the `tail` node is locked, to protect the update of its `next`

```
class LinkedList {
    ...
    public void add(Object o) {
        LinkedList obj1 = this;
        Node obj2 = this.tail;
        lockWrite(obj1);
        lockWrite(obj2); // not needed as Nodes are local to list
        Node n = new Node(o);
        if (head == null) {
            head = tail = n;
        }
        else {
            tail.next = n;
            tail = n;
        }
        unlockWrite(obj2);
        unlockWrite(obj1);
    }
}
```

Figure 5.3: The `add` method from Figure 5.1 instrumented with our inferred locks. The important observation here is that because all access to the nodes are dominated by the list, they are implicitly protected by the list's lock and so only this needs to be acquired.

field. However, because all accesses to `Node`s are internal to the `LinkedList` instance, we can protect them all with just the lock on the list, i.e. `this`.

In general, an object O_1 is instance-local to an object O_2 if O_2 creates O_1 and the only references to O_1 are made by O_2 or other objects also local to O_2. O_2 is said to form an ownership domain and it dominates all accesses to objects in this domain. This is known as *owner-as-dominator* [DM05, CDE]. All objects in this domain can be protected by simply acquiring the lock on O_2.

Dataflow Analysis

We perform a flow-insensitive escape analysis to identify instance-local objects. We now describe the basic analysis. Later, we extend it to also handle instance handover and iterators. Our escape analysis has two escape modes: *Internal* and *External* (whereby *External* < *Internal*). When an object is created, it is marked as being internal and may become external if:

- It is assigned to a field that is external.
- It is passed as an argument to a method and the receiver object is external or the method is static.
- It is returned from a non-private method.

A field may become external if:

- It is accessed through an external receiver.
- It is assigned an external reference.

Initially, static fields are marked external, instance fields are marked internal, private method parameters are internal and non-private method parameters are external. We model the return value as assignment to a special return variable `$r`, which is initially internalfor private methods and external for non-private and static methods. `this` is always internal. We model array lookups as field accesses.

Our whole-program analysis finds all reachable methods in the program (including all reachable library methods) and processes them in an arbitrary order until a fixed-point is computed. We compute per-class and per-method state during fixed-point computation. Per-class summaries keep track of the escape state of fields, while per-method summaries do so for locals, parameters and return values. Like our object access inference analysis, we analyse library methods fully.

We use the results of our escape analysis when converting the access NFA (computed during our object access inference analysis) to locks (see Section 3.3) by only inferring locks for objects that are not instance-local.

Figure 5.4 gives our transfer functions. We will now describe each function in turn. Note that we assume that the type of the left and right sides of each assignment are references.

x = y In this statement, the value of y is being assigned to x and so x and y are now aliases. Hence, `y`'s escape state should be propagated to `x`. This is the first part of the update to T_m: $x \mapsto T_m(x) \cup T_m(y)$. Furthermore, if `x`'s escape state later changes, then it means that the escape state of the object it points to has changed. As `x` and `y` are aliases (up to the point where `x` is later redefined), we should also conservatively propagate `x`'s escape state to `y`. This is safe because going from internal to external results in locking more objects. Note, this is a flow-insensitive analysis and so we have a single value for T_m for the entire method m. This means that the mapping for `x` must be the meet of all possible escape states that `x` could have in the method and that is why the meet $T_m(x) \sqcup T_m(y)$ is taken (this is also the reason why `x`'s escape state is propagated to `y`).

x = `new T` When an object is constructed, its initial escape state is internal. That is, it is considered instance-local to the enclosing object. There are certain cases when this assumption is not correct, such as if the enclosing method is `main()` or a `Thread`'s `run()` method because we assume neither to be part of an object.[1]

x = y.f This case is similar to x = y; The object referenced by `y.f` is assigned to `x`, so `y.f`'s escape state is propagated to `x` and vice-versa. It is possible that $T_c(.f)$ may be external but $T_m(y)$ internal. One option would be to update $T_m(y)$ to external, however this would then require locking `y`, despite it not escaping the enclosing object. On the other hand, we could leave $T_m(y)$ as internal and $T_c(.f)$ as external thus requiring locking `y.f` but not `y`. This still ensures soundness because all external objects are locked and is subsequently the approach we take. It is also possible that $T_m(y)$ is external and $T_c(.f)$ is internal. We take the conservative approach that if an object becomes external then all its fields also become external. This is captured by taking the meet with $T_m(y)$ when updating $T_c(.f)$. Note, the scope of internal and external are with respect to the enclosing instance.

x.f = y Similar to x = y.f.

[1] Although `run()` is an instance method of `Thread`, we have not come across a case whereby an instance field was initialised in it and thus treat all objects allocated in it as external.

x = y[*] Here, some element of the array `y` is being assigned to `x`. We abstract away the particular array index. We model array accesses as accessing a special field called `$elem` in the class corresponding to the run-time type of the element. However, unlike objects whereby the particular field being accessed is known at compile-time (i.e. which class the field belongs to), it is not possible in general to know which `$elem` field is being accessed because that requires knowing the types of array elements. For an array of type `T[]`, the type of the array elements can be `T` or any of its subclasses. We therefore use Soot's alias analysis to obtain the possible element types and then update $T_c(\$elem)$ for each of them. Again, if `y` is external, then elements also become external. As there may be many possible element types for `y`, we take the meet of all of them when updating $T_m(x)$.

x[*] = y Similar to x = y[*].

return x We treat return a value as an assignment to a special variable `$r`, i.e. `$r = x`. This variable is maintained on a per-method basis and the rest is the same as x = y.

throw x Exception objects are assumed to escape the enclosing object and are therefore assigned the escape state external.

Utility Methods

Normally, passing a local variable as an argument to a static method or an instance method of an external object makes it external. However, some methods perform a utility function, such as `arraycopy()`, which would not have affected the escape state if they had been inlined. We have found that by not treating such methods as if they were inlined, many variables and fields quickly become external. We therefore feel it necessary to treat these functions essentially as no-ops:

- `System.arraycopy(Object src, int srcStart, Object dest, int destStart, int len)`: Copy one array onto another.

$$
\begin{array}{rcl}
t_{[x=y]} & = & T_m[x \mapsto T_m(x) \cup T_m(y)][y \mapsto T_m(x) \cup T_m(y)] \\
t_{[x=\texttt{new}]} & = & T_m[x \mapsto External] \text{ (if } m \text{ is } \texttt{main()} \text{ or } \texttt{Thread.run()}) \\
\hline
t_{[x=y.f]} & = & T_m[x \mapsto T_m(x) \cup T_m(y) \cup T_c(.f)] \\
 & & T_c[.f \mapsto T_c(.f) \cup T_m(x) \cup T_m(y)] \\
\hline
t_{[x.f=y]} & = & T_m[y \mapsto T_m(y) \cup T_m(x) \cup T_c(.f)] \\
 & & T_c[.f \mapsto T_c(.f) \cup T_m(y) \cup T_m(x)] \\
\hline
t_{[x=C.f]} & = & T_m[x \mapsto External] \\
\hline
t_{[C.f=y]} & = & T_m[y \mapsto External] \\
\hline
t_{[x=y[*]]} & = & \forall c \in possibleElemTypesOf(x)\ .\ T_c[\$elem \mapsto T_c(\$elem) \cup T_m(x) \cup T_m(y)] \\
 & & T_m[x \mapsto T_m(x) \cup T_m(y) \cup \bigcup\{T_c(\$elem) \mid c \in possibleElemTypesOf(y)\}] \\
\hline
t_{[x[*]=y]} & = & \forall c \in possibleElemTypesOf(x)\ .\ T_c[\$elem \mapsto T_c(\$elem) \cup T_m(x) \cup T_m(y)] \\
 & & T_m[y \mapsto T_m(y) \cup T_m(x) \cup \bigcup\{T_c(\$elem) \mid c \in possibleElemTypesOf(x)\}] \\
\hline
t_{[x[*]=\texttt{null } or \texttt{ new}]} & = & \forall c \in possibleElemTypesOf(x)\ .\ T_c[\$elem \mapsto T_c(\$elem) \cup T_m(x)] \\
\hline
t_{[\texttt{return x}]} & = & T_m[x \mapsto T_m(x) \cup T_m(r)][r \mapsto T_m(x) \cup T_m(r)] \\
\hline
t_{[\texttt{throw x}]} & = & T_m[x \mapsto External]
\end{array}
$$

Figure 5.4: Transfer functions for instance-local object inference.

- `Arrays.fill(Object[] a, int fromIndex, int toIndex, Object val)`: Fill a range of an array with an Object value.

- `AbstractCollection.equals(Object o1, Object o2)`: compare two objects according to Collection semantics.

- `Object.clone()`: clones the object.

Iterators & Inner Classes

Iterators are commonly used to traverse collections and are usually implemented as inner classes. However, iterator instances escape their collection object and may also access its fields (e.g. `head` in `LinkedList`). With our simple analysis, these fields would consequently be tagged as external, although we have previously established that they should be internal.

We make the observation that although iterators escape, they are still logically part of the collection and should be treated as such. In particular, accessing fields of the underlying collection should not make them external. We believe this to be true of inner class instances

in general and thus extend our treatment to not just iterators but also inner classes.[2] We now describe how we do this.

When an inner class instance is created, the enclosing `this` reference is passed as the first parameter to the constructor and subsequently stored in the field `this$0`. Figure 5.5 shows an example code fragment from Java's `AbstractList` class (abbreviated to `AbsList`) to illustrate this. All accesses to the enclosing instance's fields are made through `this$0`. Usually, all parameters of a constructor are assumed to be external, however we tag this first parameter as internal. The reason for this is so that when it is assigned to `this$0` in the constructor, the escape state of `this$0` remains internal. More abstractly, we are saying that we know the reference being passed to the constructor is in the list's ownership domain. This becomes important when accessing enclosing instance fields, because field accesses through an external receiver makes the field external.

Handovers

Sometimes an object is constructed and then passed on to another object, with the creating object never accessing it. We call this a *handover*. In the example shown in Figure 5.6, an application-specific comparator instance `MyObjComparator` is created and passed to a `TreeMap` constructor. Although the comparator is created outside `map`, it is never accessed by the creating object (or any other object except `map`) and should be treated as being part of the `map`'s ownership domain. However, our current analysis marks the parameters of non-private methods as external and so is not able to do this. A handover is like a transfer of ownership [mul07].

We extend our analysis to detect handovers. We use Soot's use-def analysis to find arguments to method calls that:

1. Refer to newly constructed objects

2. Are never accessed by the creating object (except when argument passing).

[2] If an iterator is not an inner class, then our approach cannot handle them

```
class AbsList ... {
    ...
    private class Itr implements Iterator<E> {
        ...
    }
    ...
    public Iterator<E> iterator() {
        return new Itr();
    }
    ...
}
```

(a)

```
public Iterator iterator() {
    AbsList r0;
    AbsList$Itr $r1;

    r0 := @this: AbsList;
    $r1 = new AbsList$Itr;
    specialinvoke $r1.<AbsList$Itr: void <init>(AbsList,AbsList$1)>(r0, null);
    return $r1;
}

class AbsList$Itr extends Object implements Iterator {
    ...
    final AbsList this$0;
    ...
    private void <init>(java.util.AbsList) {
        AbsList$Itr r0;
        AbsList r1;
        r0 := @this: AbsList$Itr;
        r1 := @parameter0: AbsList;
        r0.<AbsList$Itr: AbsList this$0> = r1;
        ...
    }
    ...
}
```

(b

Figure 5.5: (a) Java and (b) Jimple code for `java.util.AbstractList` and its inner iterator class `java.util.AbstractList$Itr`. This example demonstrates how a reference to the enclosing `AbstractList` instance is implicitly passed to the iterator instance and stored in the `this$0` field. By ensuring that this first constructor parameter is kept internal, we trick the analysis into thinking that all fields that are marked as internal in `AbstractList` are still the case even if they are accessed by the iterator (see field access rules in Figure 5.4).

```
class TreeMap<K, V> extends AbstractMap<K, V> ... {
  final Comparator<? super K> comparator;
  ...
  public TreeMap(Comparator<? super K> c) {
    comparator = c;
    ...
  }
  ...
}
```

```
TreeMap<MyObj, String> map = new TreeMap<MyObj, String>(new MyObjComparator());
```

Figure 5.6: An example of a handover whereby a `MyComparator` object is instantiated and then passed to the `TreeMap` constructor and is never accessed again in the creating scope.

Figure 5.7 gives the initial version of our algorithm. It takes as input the set of reachable methods of the program being analysed (including all reachable library methods). For every method call *mc*, we iterate through each argument *a*. We first check that all of *a*'s reaching definitions are of the form `a = new T` (i.e. that *a* only refers to a newly constructed object). If this is the case then we check that *a* has no other live uses except *mc*. These two checks confirm that the argument is a newly created object that is not accessed by the creating scope except when passing it as an argument in *mc*.

When is a handover not a handover?

There is one subtle case when this simple algorithm would incorrectly identify a handover. The problem arises from the fact that Soot's use-def information does not distinguish between single and multiple executions of a use statement. Thus, it may return a singleton statement as the use but this statement may be executed multiple times before the variable is redefined. Figure 5.8 shows two example programs that contain loops. In Figure 5.8(a), a new object is constructed and passed to the `MyObj` constructor within the loop whereas in Figure 5.8(b) the handover object is created outside the loop and is passed an arbitrary number of times to the `MyObj` constructor. In both cases, the use-def information will return that the only use of `x` is the line `y = new MyObj(x)`, however it does not capture the fact that in the case of Figure 5.8(b), this statement is executed multiple times and so `x` is actually passed to multiple

```
findHandovers(methods) {
    for (Method m : methods) {
        for (Stmt s : m) {
            if (s is a method call) {
                args = arguments passed to call s
                for (Arg a : args) {
                    if (a is a local variable) {
                        defs = getReachingDefs(a, s);
                        if (defs are all of the form a = new T) {
                            if (no live uses of a except call s) {
                                a.handover = true
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Figure 5.7: Pseudocode of the simple version of our handover detection algorithm.

```
while (...) {
    x = new MyObj();
    y = new MyObj(x);
}
```

(a)

```
x = new MyObj();
while (...) {
    y = new MyObj(x);
}
```

(b)

Figure 5.8: Two example programs showcasing that loops can lead to incorrectly identifying a handover.

object constructors. Recall that the purpose of detecting handover arguments is so that if they are assigned to fields in the callee object, they can be treated as internal objects. Handovers can only happen once as otherwise that would constitute sharing and means these objects would need to be locked.

We update the algorithm in Figure 5.9 to detect this case. The modification looks for cycles from the method call *mc* to itself along which the argument *a* is not redefined. If such a cycle exists then we conclude that it is not a handover.

```
findHandovers(methods) {
    for (Method m : methods) {
        for (Stmt s : m) {
            if (s is a method call) {
                args = arguments passed to call s
                for (Arg a : args) {
                    if (a is a local variable) {
                        defs = getReachingDefs(a, s);
                        if (defs are all of the form a = new T) {
                            if (no live uses of a except call s
                                && a cycle from s -> s does not exist
                                    along which a is not redefined) {
                                a.handover = true
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Figure 5.9: Pseudocode of our handover detection algorithm that detects the subtle case when a prospective handover-object is passed to multiple callees and so is actually not a handover.

Allowing benign uses

Our definition of handover currently requires that the object being passed is not accessed by the creating object. However, we can relax this condition to allow the following benign uses:

- Calls to `Thread.start()` and `Thread.join()`, if the handover object is a `Thread`.
- Assignments to local variables.

In the case of `Thread` objects, it is ok to allow calls to `start()` and `join()` because the state these two methods modify are disjoint from the application-defined state in the derived class. Moreover, these methods are designed to be called by a different thread and so perform synchronisation internally. We also assume that atomic sections do not perform any threading operations whatsoever (i.e. they would not call `start()` or `join()`). Figure 5.10 shows an exemplary code fragment from the *traffic* benchmark where not making such a relaxation would otherwise prevent us from detecting a handover. A pair of `Thread`-derived `Car` and

```
1  public class Driver extends Thread {
2      ...
3      public Car car;
4      ...
5  }
6
7  public class Car extends Thread {
8      ...
9      protected Location location;
10     ...
11     private Driver driver;
12     ...
13 }
14
15 car = new Car(new Location(...), ...);
16 driver = new Driver(..., car, ...);
17 ...
18 car.setDriver(driver);    // handover point
19 Rotary.addCar(car);
20 ...
21 driver.start();
22 car.start();
23 ...
24 try { driver.join(); } catch(Exception e) { ... };
25 try { car.join(); } catch(Exception e) { ... };
```

Figure 5.10: Code fragment from the *traffic* benchmark whereby a `Driver` thread object is handed-over to a `Car` instance and later has `start()` and `join()` called on it in the creating scope.

`Driver` instances are created. The `Driver` is handed over to the `Car` before both threads are started. There are only two uses of the driver instance: (1) when being passed to the car and (2) when `start()` is invoked on it. Apart from this there are no other uses and thus the call `car.setDriver(driver)` is a handover. There is also a handover of the constructed `Location` object on line 15. Note, the `Car` instance is shared by both the `Rotary` and the driver and thus cannot be considered an instance-local object by either.

Another use that should be allowed is assignment to a local variable. Figure 5.11 shows another example code fragment. Here, an alias `o2` of `o1` is created and then passed to `o3`'s constructor. Although `o1` now has a second use statement `o2 = o1`, this is benign and does not affect it from being a handover. Care has to be taken though to ensure that not only is `o2` not used elsewhere but also that `o1` is not used, either of which would prevent the handover.

```
MyObj o1 = new MyObj();
MyObj o2 = o1;
MyObj o3 = new MyObj(o2);
```

Figure 5.11: Code fragment showing that local-to-local assignments are also benign for handover detection.

Our final extension to the handover detection algorithm incorporates allowing these two benign uses. Local-to-local copies add some additional complexity because all checks have to be extended to the resulting aliases too. Furthermore, cycle detection must now look for a definition of the form `x = new T`, whereby `x` is a or an alias of a. Figure 5.12 contains our final algorithm.

5.1.4 Class-local objects

In addition to instance-local objects, we also make the observation that sometimes objects stored in static fields do not escape the class they are created in. Such objects are typically also part of the implementation of the class but their scope spans all its instances. Recall from Section 3.3 that accessing a static field `f` in class `C` (i.e. `C.f`), requires locking `C`'s corresponding `Class` object, `C.class`.

For class-local objects, locks taken to protect accesses made within them are always dominated by locks on the defining class. So in the previous example, if we were to access state within the object `C.f`, locks on `C.class` and `C.f` would be acquired. Thus, in a similar fashion to instance-local objects, they can be protected by just acquiring a lock on `C.class`.

Figure 5.13 shows the `Rotary` class from the *traffic* benchmark. Here, there are three static fields `carsList`, `roadSegments` and `collisionDetector` that are initialised in the `initRotary` method. Furthermore, the `addCar` and `removeCar` methods add to and remove from `carsList` respectively. In this example, the objects referred to by `carsList` and `roadSegments` are never accessed outside the `Rotary` class and are therefore class-local. As a result, the atomic sections in `addCar` in `removeCar` just need to be replaced with a write lock on `C.class` and nothing more. On the other hand, `collisionDetector` is accessed by the `Driver` class and thus any accesses performed within it require locking all state inside `CollisionDetector`.

```
findHandovers(methods) {
    for (Method m : methods) {
        for (Stmt s : m) {
            if (s is a method call) {
                args = arguments passed to call s
                for (Arg a : args) {
                    if (a is a local variable) {
                        (defs, aliases) = getReachingDefsThroughCopies(a, s);
                        if (defs are of the form a = new T or x = y) {
                            if (no live uses of a or aliases except:
                                  1) call s, or
                                  2) Thread.start()/Thread.join(), or
                                  3) local-to-local copy
                                && a cycle from s -> s does not exist
                                   along which a or a local var alias is
                                   not assigned a new object) {
                                a.handover = true
                            }
                        }
                    }
                }
            }
        }
    }
}

getReachingDefsThroughCopies(l, stmt) {
    lDefs = getReachingDefs(l,stmt)
    allDefs = [ ]
    aliases = [ ]
    for each (lDef : lDefs) {
        if (lDef is of form l = m) {
            mDefs = getReachingDefsThroughCopies(m, lDef);
            allDefs += mDefs
            allDefs += [ lDef ]
            aliases += [ m ]
        }
        else if (lDef is of form l = new T) {
            allDefs += [ lDef ]
        }
        else {
            // not a handover
            return ([],[]);
        }
    }

    return (allDefs, aliases);
}
```

Figure 5.12: Pseudocode of the final version of our handover detection algorithm.

```
public class Rotary {
    ...
    static protected Vector carsList;
    ...
    static public Vector roadSegments;
    ...
    static public CollisionDetector collisionDetector;
    ...
    static public void initRotary() {
        carsList = new Vector();
        roadSegments = new Vector();
        collisionDetector = new CollisionDetector();
    }
    ...
    static public void addCar(Car car) {
        atomic {
            carsList.add(car);
        }
    }

    static public void removeCar(Car car) {
        atomic {
            carsList.remove(car);
        }
    }
    ...
}
```

Figure 5.13: `Rotary` class from the *traffic* benchmark. This class has three static fields, of which `carsList` and `roadSegments` only refer to class-local objects.

$t_{[x=y]}$	$= T_m[x \mapsto T_m(x) \cup T_m(y)][y \mapsto T_m(x) \cup T_m(y)]$
$t_{[x=y.f]}$	$= T_m[x \mapsto External]$
$t_{[x.f=y]}$	$= T_m[y \mapsto External]$
$t_{[x=C.f]}$	$= T_m[x \mapsto T_m(x) \cup T_c(f)]$ $T_c[f \mapsto T_c(f) \cup T_m(x)]$, if C is the current class $T_m[x \mapsto External]$ $T_c[f \mapsto External]$, otherwise
$t_{[C.f=y]}$	$= T_m[y \mapsto T_m(y) \cup T_c(f)]$ $T_c[f \mapsto T_c(f) \cup T_m(y)]$, if C is the current class $T_m[y \mapsto External]$ $T_c[f \mapsto External]$, otherwise
$t_{[x=y[*]]}$	$= T_m[x \mapsto External]$
$t_{[x[*]=y]}$	$= T_m[y \mapsto External]$
$t_{[\texttt{return x}]}$	$= T_m[x \mapsto T_m(x) \cup T_m(r)][r \mapsto T_m(x) \cup T_m(r)]$
$t_{[\texttt{throw x}]}$	$= T_m[x \mapsto External]$

Figure 5.14: Transfer functions for class-local object inference

Dataflow analysis

To detect such objects, we perform an analysis very similar to that for finding instance-local objects: our class-local objects analysis is also flow-insensitive and there are two escape modes: *Internal* and *External*.

When an object is created, it is marked as being internal and may become external if:

- It is assigned to an instance field.
- It is passed to a non-private instance method.
- It is passed to a static method not of the current class.
- It is returned from a non-private method.

A static field may become external if:

- It is accessed from outside the class.
- It is assigned an external reference.

Initially, static fields are marked internal, instance fields are marked external, instance method parameters are external and non-private method parameters are external. We model the return value as assignment to a special return variable `$r`, which is initially internal for private methods and external for non-private methods. `this` is always internal. We model array lookups as instance field accesses.

In general, we take a much more conservative approach with detecting class-local objects in comparison to instance-local objects. In particular, we assume that any assignments to and from instance fields cause an object to become external. We also do not handle utility methods, handovers or inner classes.

Figure 5.14 gives our transfer functions. Due to the similarity with our instance-local analysis, we do not explain all the transfer functions but rather focus on the differences.

Static field accesses Static fields become external if they are accessed from outside the class they are defined in. This is less conservative than assuming public static fields escape. Furthermore, this differs from our instance-local analysis where field accesses of local objects were still considered local. In the context of class-locality, local means "from the same class."

Instance field accesses We conservatively assume that assignments to instance fields makes an object external. This prevents us from detecting cases such as:

```
x.f = new MyObj();
C.f = x.f;
```

Not dealing with such cases simplifies the analysis tremendously, as it means we do not have to track the escape state of instance fields. Furthermore, this has been sufficient for finding many class-local objects.

Array accesses Similarly to instance field accesses, we also assume assigning to and from an array element makes an object external. Again, this prevents us from having to track the escape state of array elements and greatly simplifies the analysis.

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

Figure 5.15: Objects allocated just before atomic sections are still locked.

5.1.5 Method-Local Objects

Our lock inference analysis identifies objects that are allocated within atomic sections and does not infer locks for them. This is sound because while the atomic section is executing, these new objects are not visible to other threads. However, if the new object is allocated just before the atomic section, then we currently still lock it, despite it again not being visible yet to other threads. Figure 5.15 shows an example.

We perform an intra-procedural forwards flow-sensitive analysis to find such allocated objects. We formulate the analysis as finding objects that are method-local, but its flow-sensitive nature allows us to detect objects that are method-local at least up to the start of the atomic section (i.e. an object could escape during or after an atomic section, but this doesn't matter because we are only interested in what is method-local at the start of the atomic section). Our dataflow analysis propagates sets of variables that are found to escape the method. Escaping could be caused by a method call, assignment of another escaping value, assignment of/to a static field, returning from a method or throwing an exception. Figure 5.16 gives our transfer functions. We now describe the interesting transfer functions in turn:

`x = y` In this statement, the value of y is being assigned to the value of x and so x and y are now aliases. Hence, if y escapes then so should x and our transfer function adds x to the input set s appropriately. Note that x is first killed from s, as its value is being overwritten.

x = y.f Our analysis is very conservative and we do not track the escape state of fields and thus just assume the worst case that they escape a method. This is reflected by unconditionally adding x to the set. Note that we found this level of conservatism fine for detecting all new object allocations that occur just prior to an atomic section in the programs we have looked

$$
\begin{array}{rcl}
t_{[x=y]} & = & \lambda s.s \setminus \{x\} \cup \{x \mid y \in s\} \\
t_{[x=\texttt{new } or \texttt{ null}]} & = & \lambda s.s \\
\hline
t_{[x=y.f]} & = & \lambda s.s \cup \{x\} \\
t_{[x=C.f]} & = & \lambda s.s \cup \{x\} \\
\hline
t_{[x.f=y]} & = & \lambda s.s \cup \{y \mid x \in s\} \\
t_{[C.f=y]} & = & \lambda s.s \cup \{y\} \\
t_{[x.f=\texttt{new } or \texttt{ null}]^n} & = & \lambda s.s \\
\hline
t_{[x=y[*]]} & = & \lambda s.s \cup \{x\} \\
\hline
t_{[x[*]=y]} & = & \lambda s.s \cup \{y\} \\
\hline
t_{[x[*]=\texttt{new } or \texttt{ null}]} & = & \lambda s.s \\
\hline
t_{[\texttt{return x}]} & = & \lambda s.s \cup \{x\} \\
\hline
t_{[\texttt{throw x}]} & = & \lambda s.s \cup \{x\} \\
\hline
t_{[x=y.m(a_1,...,a_n)]} & = & \lambda s.s \cup \{x, a_1, ..., a_n\}
\end{array}
$$

Figure 5.16: Transfer functions for our method-local objects analysis. The analysis tracks which variables refer to objects that may escape the method.

at. If necessary, the precision could be improved by maintaining per-field escape states on a class-wide basis (as if a field escapes in one method then it has escaped in all methods).

x.f = y As mentioned previously, we assume that fields always escape a method. However, if the receiver object does not escape, then no other method has a reference to it and thus no other method can access the field yet. Hence, we only add y to the set if x escapes.

5.1.6 Dominators

So far we have shown how we identify instance- and class-local objects and avoid locking them. The reason behind this is that all accesses to these objects are dominated by their enclosing instance or class respectively. We now generalise this idea to find all locks dominated by some lock: A lock L1 dominates another lock L2 if whenever L2 is acquired then so is L1. This is clearly a generalisation of the aforementioned analyses because in those scenarios, the lock on the enclosing object or class is always acquired when the internal objects are locked.

Figure 5.17 shows an example of this more general notion of domination, comprising three atomic sections. `x`, `y`, `z` and `a` are shared objects constructed in lines 1-4. The locks our current

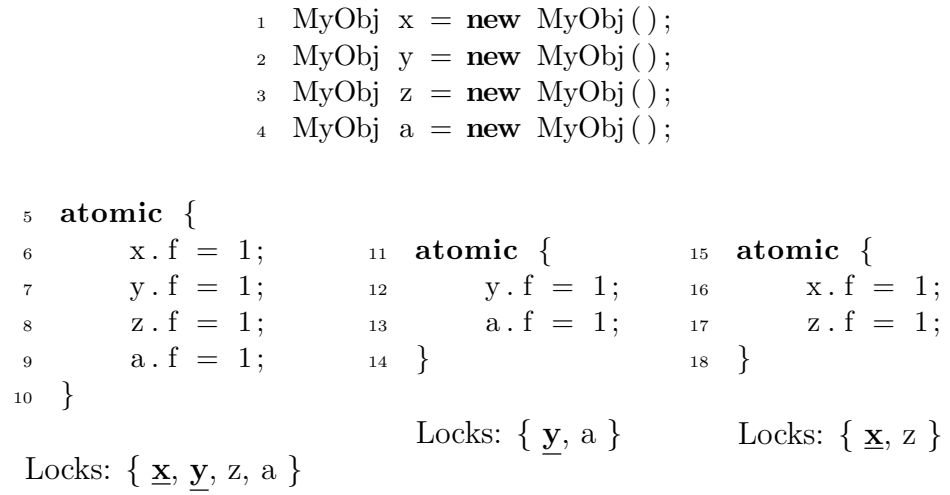


Figure 5.17: Example demonstrating the concept of dominator locks. Here, we have three atomic sections that each access two of the shared objects `x`, `y`, `z` and `a`, with the locks inferred for each atomic section written below it. We see that `x` dominates `z` and `y` dominates `a`. The dominated locks do not need to be acquired. The final set of locks to take are underlined.

lock inference analysis infers are shown below each atomic section. We see that `x` dominates `z` and `y` dominates `a`. As a result, neither `z` nor `a` need to be acquired and we can remove them. The final locks that should be taken are underlined. Notice again how the dominated objects are thread-shared but they are implicitly protected by another lock, in this case `x` for `z` and `y` for `a` so we can avoid locking them.

We now present our analysis for identifying dominated objects.

Dataflow analysis

In order to be able to perform this analysis, we need to know which locks are taken by each atomic section. This requires firstly knowing which objects are accessed. However, we infer path expressions which although resolve to concrete objects at run-time, do not tell us anything at compile-time about which objects these are. For example, the first atomic section in Figure 5.17 accesses the paths `x`, `y`, `z` and `a`, however we do not know just from the paths alone whether they refer to the same or different objects. We therefore need to employ Soot's points-to analysis to map path expressions to abstract objects.

One complication that arises with abstract objects is that they could correspond to several

objects at run-time. This happens when the associated allocation site is executed multiple times. Thus, even if we find that the abstract object $\hat{o1}$ is always locked when abstract object $\hat{o2}$ is, i.e. that $\hat{o1}$ dominates $\hat{o2}$, written $\hat{o1} \sqsupseteq \hat{o2}$, it might be the case that at run-time this doesn't hold (because they may correspond to different pairs of concrete objects on different executions). However, if we can show that $\hat{o1}$ refers to only a single unique run-time object, then that would mean that the same lock was acquired regardless of what object $\hat{o2}$ was at run-time. Hence, a requirement for $\hat{o1}$ in $\hat{o1} \sqsupseteq \hat{o2}$ is that $\hat{o1}$ can only resolve to a single unique run-time object.

Figure 5.18 gives our algorithm for finding dominator and dominated locks. We now describe the different stages involved. Note that for ease, we first calculate which abstract objects are dominated and then use this information to find dominated locks. Hence, all data structures used by our analysis store abstract objects and not locks. The algorithm starts by assuming that abstract objects that resolve to a single run-time object dominate all other objects. We employ Soot's built-in *run-once-run-many* analysis that determines for each program statement whether it is executed once or multiple times. We then use this to check if an allocation site is only executed once. If so, then that allocation site only creates a single object at run-time and we treat it as a potential dominator. All remaining objects are initially assumed to be potentially dominated by these dominators. Building this initial approximation is shown in lines 4-20. The `dominatedToDominators` relation maps an object to the set of objects that dominate it.

The next stage is to refine this initial approximation, repeatedly removing invalid dominators until we reach a fixed point. Recall that L1 dominates L2 if whenever L2 is locked then so is L1. Thus, we iterate through each object that is locked for each atomic section. We find out what its current set of dominators are and check if they are all also locked by the current atomic section. Those that are not are removed from the set. This process continues until there are no more changes to the `dominatedToDominators` relation. This step is shown in lines 22-36.

It is possible that an object may have multiple dominators. Thus, the third stage, shown in lines 38-43, is to map each dominated object to a single dominator. For example, object O may

have dominators D1 and D2. This means that D1 and D2 will both be locked whenever O is. We could leave them as is, but by identifying only one as the dominator of O, we leave open the possibility that the others may also be dominated. The `dominatedToDominator` relation maps each dominated object to its single dominator.

The final step of the algorithm is to use the information computed about which abstract objects are dominated to determine the dominated locks. A path expression may resolve to multiple abstract objects, however, as long as all of them are dominated we can conclude that the path expression will point to a dominated object and thus the corresponding lock does not need to acquired. This step is shown in lines 45-51.

Read/write locks

Our lock inference analysis makes a distinction between read and write locks. This adds a complication to the dominator analysis if a write lock *wl* is dominated by a read lock *rl*. Given that *wl* will not be acquired, race conditions may ensue. To rectify this, we must upgrade the dominator *rl* to a write lock. We extend the algorithm in Figure 5.19 to perform this.

We first find out for each object whether it is ever write locked. This is stored in the relation `objectToWrite`. We then build a mapping from dominators to the objects they dominate and iterate through it to find read-locked objects that dominate write-locked ones. These are the dominators that need to be upgraded. Finally, we find the locks corresponding to these dominators and replace them with write locks.

5.1.7 Read-only locks

We distinguish between read and write locks, to allow multiple readers of an object to be able to proceed in parallel. However, if an object is never written to, then acquiring even a read lock is unnecessary. The same also applies to type locks: if a type lock is only ever acquired in read mode, then it does not need to be locked at all. The interesting bit comes when we consider the cross dependencies between types and instances due to the multi-granularity locking protocol.

```
1  dominatedToDominators : AbsObject -> P(AbsObject);
2  dominatedToDominator : AbsObject -> AbsObject
3
4  // Step 1: Build initial approx. of dominated -> dominators relation.
5  // Potential dominators are those abstract objects that refer to a
6  // single unique run-time object. All objects are initially dominated
7  // by all potential dominators.
8  dominatedToDominators = { }
9  for each atomic section a {
10     potentialDominators = potentialDominated = { }
11     for each lock l of a {
12         objs = pointsToSetOf(l);
13         add objs to potentialDominated
14         if (objs is a single unique object o)
15             add o to potentialDominators
16     }
17     for each obj o in potentialDominated {
18         add potentialDominators to dominatedToDominators[o]
19     }
20 }
21
22 // Step 2: Fixed point calculation. Iterate through each atomic section
23 // and each obj and remove invalid dominators from dominatedToDominators
24 while there is a change {
25     for each atomic section a {
26         objs = pointsToSetsOf(locks of a)
27         for each dominated -> dominators mapping in dominatedToDominators {
28             for each d in dominators {
29                 if d is not in objs {
30                     // d is not acquired in this atomic
31                     remove d from dominators
32                 }
33             }
34         }
35     }
36 }
37
38 // Step 3: Each lock is dominated by at most one dominator lock
39 dominatedToDominator = { }
40 for each dominated -> dominators mapping {
41     dominator = pick first dominator in dominators
42     dominatedToDominator[dominated] = dominator
43 }
44
45 // Step 4: Mark dominated locks
46 for each atomic section a {
47     for each lock l in a {
48         objs = pointsToSetOf(l)
49         l.dominated = areAllObjsAreDominated(objs)
50     }
51 }
```

Figure 5.18: Algorithm for finding dominators.

```
//
// Step 5: If a read lock dominates a write lock, the
// dominator should be upgraded to a write lock.
//

// Step 5a: build Object -> isWrite relation
objectToWrite : Object -> Boolean
objectToWrite = { }
for each atomic section a {
    for each lock l in a {
        objs = pointsToSetOf(l)
        for each obj o in objs
            objectToWrite[o] |= l.isWrite();
    }
}

// Step 5b: build dominator -> dominated relation
dominatorToDominated : Object -> P(Object)
dominatorToDominated = { }
for each (dominated, dominator) in dominatedToDominator
    add dominated to dominatorToDominated[dominator]

// Step 5c: find dominators that are only read but that dominate objects
// written to
dominatorsToUpgrade = { }
for each dominator -> dominated mapping in dominatorToDominated {
    if (!objectToWrite[dominator]) { // dominator is only read
        for each d in dominated {
            if (objectToWrite[d]) {
                // dominated object is write locked
                add dominator to dominatorsToUpgrade
            }
        }
    }
}

// Step 5d: find the corresponding dominator locks that need to
// be upgraded.
for each atomic section a {
    for each lock l in a {
        objs = pointsToSetOf(l)
        if non-empty intersection of objs with dominatorsToUpgrade {
            upgrade l to a write lock
        }
    }
}
```

Figure 5.19: Extension to our basic algorithm for finding dominators (see Figure 5.18) that handles read locks dominating write locks. In this case, the dominator must be upgraded to a write lock to prevent race conditions from ensuing.

In particular, a read-only object not only requires that it is never write locked but also that its type isn't either. Similarly, a read-only type lock additionally requires that none of its instances are ever write locked. We present our algorithm for finding read-only instance and type locks in Figure 5.20. It consists of two steps, which we now describe.

Knowing whether a write lock is ever acquired on an object or type requires looking across all atomic sections. Recall from our dominators analysis that we need to use abstract objects to approximate what path expressions resolve to at run-time. These abstract objects also give us the corresponding run-time type of the object allowing us to perform the cross-dependency checks described above. The first step of our analysis iterates through each lock of each atomic section and records which abstract objects and types are write locked. This is stored in the `objectToWrite` and `typeToWrite` maps respectively.

Having identified what is write locked, we can then proceed to actually find which locks are read-only. In the second step of the analysis, we again iterate through each lock l of each atomic section. If l is an instance lock, we check that none of the abstract objects it could resolve to or their corresponding types are write locked. On the other hand, if l is a lock on type t, we check that t and its instances are only ever read-locked.

5.1.8 Unnecessary intentional locking

We use the multi-granularity locking discipline of Gray et al [GLP75] to simultaneously support both instance locks and type locks. Before attempting to acquire an instance lock, the corresponding type lock must be acquired in intentional mode. This is a request to the type lock that, locking is being performed on one of it's instances, is it ok to proceed? If the type lock has already been acquired then the request will be refused or the thread will block. Once accepted, an attempt to lock the instance can then be done with blocking being performed again if it is not available. This protocol ensures lock arbitration between types and instances and is the crux of how multi-granularity locking works. However, notice that if a type lock is never acquired then it's instances don't first need to check whether it has been acquired in a conflicting mode or not. Lock requests on an instance can immediately proceed to try

```
1  objToWrite : AbsObject -> Boolean
2  TypeToWrite : Type -> Boolean
3
4  // Step 1: build objToWrite and typeToWrite relations
5  objToWrite = { }
6  TypeToWrite = { }
7  for each atomic section a {
8      for each lock l in a {
9          if l is a path lock {
10             objs = pointsToSetOf(l)
11             for each obj in objs {
12                 objToWrite[obj] |= l.isWrite()
13             }
14         } else { // l is a type lock
15             type = get type locked by l
16             typeToWrite[type] |= l.isWrite()
17         }
18     }
19 }
20
21 // Step 2: find instance and type locks that are read-only
22 for each atomic section a {
23     for each lock l in a {
24         if (l is a path lock) {
25             objs = pointsToSetOf(l);
26             if (no obj in objs is write locked) {
27                 types = runtimeTypesOf(objs)
28                 if (no type in types is write locked) {
29                     l.readOnly = true;
30                 }
31             }
32         }
33         else {
34             type = get type locked by l
35             boolean readOnly = !typeToWrite[type];
36             if (readOnly) {
37                 for each obj in objToWrite.keys {
38                     if (objToWrite[obj]) {
39                         objType = get run-time type of obj
40                         if (objType == type) {
41                             readOnly = false;
42                         }
43                     }
44                 }
45             }
46             l.readOnly = readOnly;
47         }
48     }
49 }
```

Figure 5.20: Algorithm for finding read-only instance and type locks.

and acquire the instance lock. We statically identify when this is the case and elide intentional locking for it. Our analysis maps each instance lock to all possible run-time types of the objects it could resolve to and check if any of those types are ever locked.

5.1.9 Lock elision for single-atomic execution

In Section 5.1.1 we described how we dynamically elide locks when only a single application thread currently exists. An extension of this is to elide locks when only one thread is executing inside an atomic section. As we only guarantee weak isolation, locks need to be taken just when multiple atomics are executing in parallel. Thus, if we know that only one thread is currently executing inside an atomic, we do not need to acquire any locks. However, care has to be taken because a thread T2 could enter an atomic section while the current thread T1 is executing with locks elided. In this case, T2 would have to wait until T1 exited its outermost atomic section. This could be implemented using an additional read/write lock that is normally acquired in read mode but is acquired in write mode before locks are elided.

We have not implemented this optimisation but feel that it would benefit workloads that mostly perform local computation and have a small shared portion as well as workloads that are irregular.

5.2 Lock implementation

Having presented analyses to reduce the number of locks inferred, we now look at the performance of our locks themselves. This is important because the conservative nature of lock inference means that the number of inferred locks will inevitably be high. Each lock acquisition and release adds additional overhead so it is important to make them as efficient as possible. Furthermore, most production-quality lock implementations are highly optimised [BKMS98, RD06, Lea05], so for lock inference to be able to compete against hand-crafted locking, we must ensure that our locks are as fast as possible.

5.2.1 Multi-granularity locking protocol

Before describing our optimised lock implementation, it is necessary to first understand how they abstractly work. Recall that our lock inference approach uses the multi-granularity locking discipline of Gray et al [GLP75] to have both instance locks and type locks.

To illustrate how multi-level locks work, we present a modified version of the famous bank account example comprising a bank having a number of branches, which in turn have a number of accounts. Figure 5.21 shows an example bank H, in which there are two branches b_1 and b_2 with three accounts each. We assume that each node in the graph has a lock associated with it. If we were using normal single-level locks and wished to sum the balances of all accounts in branch b_2, we would first acquire a read lock on H, followed by branch b_2 and finally on all account objects in b_2 (a_4, a_5 and a_6). The summation operation should be atomic and so all accounts must be locked to prevent concurrent modifications, however if the number of accounts is large this will result in a lot of lock acquisitions.

What is actually occurring here is that data are being accessed at the granularity of a branch. Multi-level locks allow the entire branch including all its accounts to be locked by acquiring only b_2's lock. Note, acquiring the multi-level lock on an account has the same behaviour as in the single-level lock case (as there are no child nodes). In general, multi-level locks can be acquired in either *shared* (S) or *exclusive* (X) mode, each of which implicitly locks all child nodes in the same mode.

Given the hierarchical nature of multi-level locks, care has to be taken to ensure that an ancestor node hasn't already been locked in a mode that is incompatible (e.g. trying to acquire the S lock on b_1 when H's X lock has already been acquired by another thread). To prevent this, two additional modes are used: *intention shared* (IS) and *intention exclusive* (IX), which indicate that S or X locking is to be performed respectively further down the graph. For example, before acquiring the S lock on b_2, the IS lock has to be acquired on H. As another example, suppose we wished to perform a deposit on account a_5 and thus required acquiring a_5's X lock. In this case, we would first acquire the IX lock on H, then the IX lock on b_2 and then the X lock on a_5.

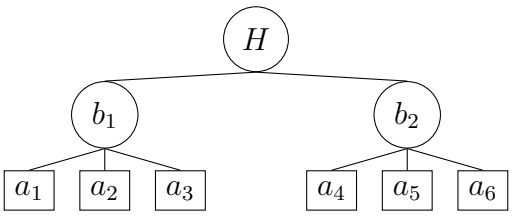


Figure 5.21: Bank account example structured into multiple branches

Figure 5.22(a) gives the partial ordering of the different lock modes and Figure 5.22(b) shows which modes can be simultaneously granted to distinct threads.

Note, an additional mode called *Shared Intention Exclusive* (SIX) is also used to achieve more concurrency in the common case where a thread may read many nodes in a sub-tree but only write to a few. Normally, the thread would need to acquire the X lock on the sub-tree but this is overly conservative, as it prevents concurrent threads from performing reads lower down. Please refer to [GLP75] for the full details of multi-level locks.

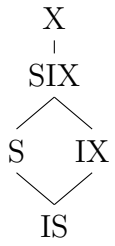
We now describe our optimised implementation of these locks.

5.2.2 The Synchronizer Framework

In Java, production-quality locks are built using the Synchronizer framework [Lea05]. This is part of the `java.util.concurrent` Java Concurrency library and provides common mechanics for atomically managing synchronisation state, blocking and unblocking threads, and queuing. Synchronisation state is represented using a single 32-bit integer value and queues are non-blocking. All state updates are performed using CAS. All these behaviours are encapsulated in the base class `AbstractQueuedSynchronizer`[3]. AQS internally supports the two modes *exclusive* and *shared*, however the framework is flexible as to how a custom synchroniser's specific modes map to them. To implement a custom synchronizer, AQS is extended and the `tryAcquire`, `tryRelease`, `tryAcquireShared` and `tryReleaseShared` methods are overriden.

Multi-level locks have five modes they can be acquired in: *exclusive* (X), *shared* (S), *intention shared* (IS), *intention exclusive* (IX) and *shared intention exclusive* (SIX) [GLP75]. Note,

[3]There is also a `long` version, called `AbstractQueuedLongSynchronizer`, that uses 64-bits for state



(a)

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

(b)

Figure 5.22: (a) Mode lattice and (b) compatibility matrix (from [GLP75]). The compatibility matrix shows which modes can be acquired concurrenty by different threads.

SIX can be implicitly represented by non-zero counts for both S and IX, hence we only explicitly represent the four modes X, S, IS, IX allocating 16 bits for each of their counts (we use `AbstractedQueuedLongSynchronizer`). This allows up to 65535 reentrant acquires in each mode. We map X to exclusive and the remaining three modes (S, IS, IX) to shared. The disambiguation of the latter three modes is made in the `tryAcquireShared` and `tryReleaseShared` methods.

We also extend the `Thread` class to store how many times the current thread has acquired the lock in each mode. This makes thread-local lookups much faster than using `ThreadLocal`, as the latter performs a hash table lookup.

5.3 Deadlock

The final cause of runtime overhead we found was due to our deadlock-avoidance scheme. Every time an acquisition on some lock l fails, we essentially rollback the locking phase and re-acquire all locks. We do this by releasing all already acquired locks that precede l before blocking and waiting for l to become available. When this eventually occurs, we immediately release l and then reacquire all locks from the start. We also employ an exponential backoff to minimise the chance of livelock from occurring. Section 3.4 describes our approach. However, the overhead that arises from blocking until l becomes available and for the backoff can be costly. Furthermore, much of the time locks become available shortly after `tryLock` returned false. Thus, rather than being overly conservative and assuming that a deadlock may have

```
1 for (int i=0; i<MAX_POLL; i++) {
2     if (l.tryLock()) {
3         ... continue lock acqusitions ...
4     }
5     for (int j=0; j<WAIT_BETWEEN_POLLS; j++) { }
6 }
```

Figure 5.23: When a lock is not available, we poll it a few times first before rolling back the locking phase.

occurred at the first failure to acquire a lock, we poll the lock a few times first before rolling back. Figure 5.23 shows our loop for polling on l. This adaptive scheme has improved performance tremendously, as we shall see in our evaluation.

5.4 Evaluation

We now evaluate our various optimisations for reducing the number of locks, our optimised lock implementation and our improved deadlock-avoidance algorithm. So far, we have been able to analyse very large amounts of code but the resulting performance has been very poor. The motivation of this chapter has been to find techniques to improve this performance. We now show that through the optimisations listed in this chapter, we have been able to get performance very close to that of the original hand-crafted locks.

Due to our analysis optimisations in Chapter 4, we can now analyse our benchmarks on the commodity machine *liatris*. However, hsqldb's memory requirements remain high so we still analyse it on *ax3*. Furthermore, we assume the use of the optimised lock implementation and deadlock-free lock acquisition loop.

Figure 5.24(a) shows the number of locks inferred by our analysis both with and without all the lock optimisations from Section 5.1. Moreover, Figure 5.24(b) shows a breakdown of how many locks are reduced by each individual optimisation.

Our optimisations are successful in significantly reducing the number of locks. For example, in the case of *hsqldb* there is a 75% reduction and in the case of *mtrt* 94%!

Program	(i) Halpert		Ours							
			(ii) No lock opt.				(iii) With all lock opt.			
	Static	*Dynamic*	*Inst.*		*Type*		*Inst.*		*Type*	
			R	W	R	W	R	W	R	W
sync	0	2	1	2	0	0	0	2	0	0
pcmab	0	3	1	5	0	0	0	2	0	0
bank	0	3	0	12	0	0	0	6	0	0
traffic	0	19	33	67	0	0	11	18	0	0
mtrt	1	0	905	268	726	130	0	48	6	66
hsqldb	2	11	32508	24956	26429	10943	1725	4155	9792	8301

(a)

Program	(i) TLO				(ii) ILO				(iv) CLO		(vii) MLO				(iii) DOM		(v) RO				(vi) IMP	
	Inst.		*Type*		*Inst.*		*Type*		*Inst.*		*Inst.*		*Type*		*Inst.*		*Inst.*		*Type*		*Inst.*	
	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W
sync	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	2
pcmab	0	1	0	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5
bank	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	12
traffic	0	1	0	0	4	41	0	0	1	6	0	2	0	0	1	0	31	0	0	0	31	49
mtrt	52	5	24	20	92	57	24	60	119	6	0	0	0	0	491	204	613	0	702	0	560	63
hsqldb	464	6045	492	450	2352	3315	1682	2552	4951	487	0	0	0	0	19775	13780	17948	0	15672	0	15070	2276

(b)

Figure 5.24: Locks inferred for benchmarks in Figure 3.19 by Halpert et al (a)(i) and our approach for both without (a)(ii) and with all our lock optimisations enabled (a)(iii). (b) gives a breakdown of how many locks are removed by each optimisations.

Program	**Lock optimisations (secs)**						
	TLO	ILO	CLO	MLO	DOM	RO	IMP
sync	0.598	8.441	3.979	0.0010	1.42	0.0010	0.0
pcmab	0.603	8.309	3.855	0.0020	1.444	0.0010	0.0
bank	0.408	8.177	3.802	0.0020	1.376	0.0020	0.0010
traffic	0.569	9.267	3.861	0.465	1.625	0.0060	0.0020
mtrt	0.623	9.063	4.259	0.0050	1.741	0.079	0.03
hsqldb	1.667	28.589	53.125	0.079	9.597	1.84	2.724

Figure 5.25: Analysis time breakdown for each lock optimisation.

In the breakdown, we find that the optimisation that removes the most number of locks varies between the benchmarks. However, for benchmarks with very large numbers of locks it appears that the dominators analysis seems to be most successful. The reason for this might be because of large numbers of common code paths and thus large numbers of common locks between atomic sections.

Most impressive are the run times shown in Figure 5.26. We see that by reducing the number of locks, the run-time performance has drastically improved. Most notably, is that of hsqldb: from 160x slower to just 3.5x slower - a improvement factor of 45. Furthermore, we see slight speed ups in the *sync* and *bank* benchmarks.

Program	Run-time (secs)			
	Manual	Global	Halpert	Ours
sync	69.14	71.22	72.69	56.61
pcmab	2.28	3.15	2.28	2.47
bank	20.89	19.50	35.69	3.88
traffic	2.56	4.22	2.65	4.42
mtrt	0.80	0.82	0.78	0.85
hsqldb	3.25	3.12	3.25	11.39

Figure 5.26: Comparison of execution times for each benchmark. The times we report for our approach are with all lock optimisations enabled.

5.5 Conclusion

Despite having a highly scalable analysis, we found that the number of locks we were inferring were too high. This is because we assumed that all object accesses need to be locked. This negatively impacted performance as we had slowdowns of upto 160x on the hsqldb benchmark.

The reason for the bad performance was due to three things: too many locks being inferred, an inefficient lock implementation and too much blocking when acquiring locks. In this chapter, we have dealt with each of these and have presented several techniques to deal with them.

To reduce the number of locks, we have presented analyses for identifying thread-local, instance-local, class-local, method-local, dominators and read-only locks. We also identify when it is not necessary to acquire multi-granularity locks in intentional mode and when it safe to elide locks completely. Our optimised multi-granularity lock implementation is built using Java's Synchronizer framework and we reduce context-switch overhead in our locking acquisition code by polling locks for a short period.

Our analyses are very fast and our results show that we gain up to 94% reduction in the number of locks. More impressive is the execution time improvements of the resulting instrumented programs. We obtain performance very similar to the original hand-crafted version of the benchmark. In the case of hsqldb, our performance improves from a 160x slowdown to now just a 3.5x slowdown.

What we have achieved is a sound, scalable analysis that is able to handle large Java programs making use of libraries and containing atomic sections involving I/O and system calls, that

infers a reasonably efficient set of locks whose resulting performance is close to the original program! This is the first lock inference approach to achieve this.

Chapter 6

Conclusion

6.1 Summary of thesis achievements

6.1.1 Recap of motivation

Atomicity provides strong guarantees against errors caused by unanticipated thread interactions. However, manually enforcing atomicity is error-prone and sometimes not even possible. As a result, a programming abstraction has been proposed, called *atomic sections* that allow a programmer to declaratively mark a block of code as executing atomically and leave the details of how this is achieved to the compiler and/or run-time.

Atomic sections are a language level abstraction and thus the question of how to actually implement their semantics has been a very important research question for the last 10-15 years. Transactional memory has been the most popular technique, in which memory updates are buffered during executed and committed in a single step at the end provided no conflicting updates have been performed by a concurrent thread. If there has been, then the buffered updates are discarded and the atomic section is re-executed. While it has the advantages of scalable performance, transactional memory suffers from high execution overhead due to logging and roll back but more importantly is unable to handle irreversible operations such as I/O and system calls. As a result, expressivity of atomic sections is called into question.

Lock inference is an alternative technique that statically infers the locks that need to be acquired to ensure atomic execution and instruments them into the program. It is pessimistic in nature, as locks are acquired on shared data before the shared data is accessed, but this enables it to support irreversible operations. That is the reason why we have decided to pursue lock inference in this thesis. However, although expressivity is now restored, we have found through the very simple Hello World atomic section (see Section 1.6) that even small programs rely on very large parts of the library. Thus, for a lock inference approach to be able to handle even small real-world programs, it needs to be able to scale to the library. This is problematic because all prior work has shied away from tackling the library. They either (i) ignore it, (ii) requires library implementors to annotate which locks to take or (iii) analyse library call chains only upto one-level deep. All of these approaches may result in accesses remaining unprotected.

The reason why lock inference has avoided libraries is because they are notoriously known to be a challenge for static analysis, due to (i) their high cyclomatic complexity, (ii) their generality and (iii) the lack of available source code for them. Therefore, the motivation for this thesis is to tackle this scalability problem and more generally argue the following:

"It is possible to develop lock inference techniques that scale to real-world Java programs that make use of the library and still obtain performance comparable to hand-crafted locking."

6.1.2 Achievements

We believe that this is the first lock inference approach that can analyse precisely Java programs built with large libraries and achieve performance similar to that of hand-crafted locking:

- We are able to handle library programs by formulating our previous path inference analysis [CGE08] as an IDE dataflow problem. We refine the pointwise representations of [SRH96, RSX08] and show that our analysis can scale to the entire GNU Classpath library (122KLOC).
- We have presented a number of analysis optimisations, namely: CFG summarisation, delta propagation, worklist ordering and parallel propagation, which in turn we have

evaluated. These optimisations enable us to scale to very large code bases such as the large Java database engine HSQLDB comprising 150KLOC (plus 3000 methods from GNU Classpath). Most notable of these optimisations is our novel delta transformers that dramatically reduce analysis time and memory requirements.

- We have also implemented several analyses to identify and eliminate locks inferred for: thread-local, instance-local, class-local, method-local, dominated and read-only objects. We also dynamically elide locks for atomic sections when there is only a single thread executing in the application. All our analyses are completely automatic and do not require any programmer-annotations. Our lock inference approach is the first to automatically identify instance- and class-local objects as well as the more general notion of dominated locks and elide them at compile-time. Furthermore, these analyses are conservative but scale to library code and are still able to identify many such objects. We evaluate their effectiveness on a suite of benchmarks. We also present an efficient implementation of Gray et al [GLP75]'s multi-granularity locks using Lea's Synchronizer framework [Lea05]. Finally, we optimise our deadlock-free lock-acquisition loop by polling locks for a short while prior to blocking on them.

- We present a full implementation of all our analyses in the Soot framework and evaluate them on the motivating Hello World program as well as GNU Classpath and a suite of benchmark programs. We show that with our techniques, we are able to achieve performance very close to hand-crafted locks with a maximum of 3.5x slowdown on hsqldb. This is an important achievement, as we provide the programming model of using a single global lock, but performance that is close to expert hand-crafted locks. We compare results with Halpert et al [HPV07]. They only analyse library call chains up to one level deep. For benchmarks that involve little library code, we obtain similar performance but for programs that make extensive use of the library, we are slower. However, our approach analyses all library code and is therefore sound, whereas it can be shown that Halpert et al's approach can produce unsound results (see Section 2.5.2).

6.2 Future work

In this section, we identify possible future directions of work.

6.2.1 Cold code paths

Lock inference relies heavily on static analysis to identify a suitable set of locks for atomicity. As static analysis is done at compile-time, the results it computes must cover all possible executions of the program to ensure safety. Some program paths are not executed very frequently because they may cover special cases (e.g. exception handlers), however, static analysis has to conservatively assume that because there is a possibility that they can be executed, locks must be inferred to protect their accesses. However, these locks are only required in certain situations and thus acquiring them on every execution of the atomic section adds unnecessary lock contention and overhead.

One area of future work is to differentiate such *cold code paths* from the frequently executed ones and defer acquiring their locks until absolutely necessary. So, locks for normal code could be acquired as usual at the start of the atomic section but locks protecting accesses made along cold code paths would be acquired at the start of the cold region. Figure 6.1 shows an example to illustrate how this could work. Figure 6.1(a) shows a try-catch block inside an atomic section. Currently, our analysis would infer the locks as shown in Figure 6.1(b), however note that the lock on `y` is only necessary if the catch block is executed, which is very rare. What we propose is to treat the exception handler as cold code and thus defer locking `y` to the point where it is executed, i.e. the start of the catch block. This still ensures safety as `y` is locked before being accessed but it reduces the number of locks that are initially acquired. The resulting locking policy is shown in Figure 6.1(c).

This kind of technique is especially useful when library code is involved, because the library is specifically designed to be general purpose and cater for many possible usage contexts, many of which are not executed frequently. For example, the character set loading code that is executed when a string is printed (see the Hello World example in Section 1.6) for the first

```
atomic {
    try {
        x.f = 1;
    }
    catch (Exception e) {
        y.f = 10;
    }
}
```

(a)

```
lockWrite(x);
lockWrite(y);
try {
    x.f = 1;
}
catch (Exception e) {
    y.f = 10;
}
finally {
    unlockWrite(x);
    unlockWrite(y);
}
```

(b)

```
boolean yLocked = false;
lockWrite(x);
try {
    x.f = 1;
}
catch (Exception e) {
    lockWrite(y);
    yLocked = true;
    y.f = 10;
}
finally {
    unlockWrite(x);
    if (yLocked) {
        unlockWrite(y);
    }
}
```

(c)

Figure 6.1: Example illustrating the concept of cold code paths and how they can utilised to optimise the locking policy.

time, does not execute the second time. Thus, on the second execution of the atomic section, those corresponding locks do not need to be acquired. However, due to the conservative nature of static analysis all these accesses have to be locked.

The conservativeness of static analysis is a universal problem and we believe that differentiating cold code from common code paths is one useful way to improve the precision of analysis results without losing soundness. How this soundness is maintained for cold code is analysis-dependent, but in our case it would mean acquiring cold locks at the start of those code regions. To the best of our knowledge, such a distinction has never been considered by prior work on static analysis.

Identifying cold code paths

Cold code paths can be identified using profiling information about which bytecode instructions and methods are executed. Care has to be taken to make sure that a reasonably extensive set of inputs are used to prevent mistaking a common code path for a cold one. In a preliminary

exploration of this, we integrated the Emma[1] code coverage tool into JikesRVM so that we could get a list of the methods that were called while the application executed. Although JikesRVM is able to provide this information, emma gives very nice output in the form of html files and maps the bytecode execution information back to the corresponding high-level Java statements, making it easier to review. The results we obtained were very encouraging. For example, we found that for hsqldb, only 2745 out of its 5062 call graph methods (54%) are executed!

Deadlock

By deferring acquisition of locks for cold code paths, locks are no longer all acquired at the start of the atomic section. This means we can no longer simply roll back the locking phase, as shared memory updated may have been performed.

To avoid deadlock, an analysis could be performed to identify which cold locks may be involved in deadlocks and then push those specific locks to the top of the atomic section. Although some cold locks will therefore be acquired on every execution, we believe that the number of deferred locks will still be high. Figure 6.2(a) shows the try-catch atomic from Figure 6.1 together with another conflicting try-catch atomic. We treat both exception handlers as cold code paths and the resulting locking policy is shown in Figure 6.2(b). However, note that now locks `x` and `y` are acquired in reverse orders and consequently a deadlock could result. To remedy this we acquire them at the start of each atomic section, but as deferring the lock on `z` is still fine, we leave its acquisition to the start of the catch block. The final fixed version is shown in Figure 6.2(c).

6.2.2 Eliminate type locks

Our lock inference approach uses instance locks whenever possible and type locks for when our analysis infers a statically unbound set of accesses. Acquiring a lock on type t implicitly acquires locks on all of t's instances. This can be too coarse as in reality only a fraction of these

[1] Available from http://emma.sourceforge.net

```
atomic {
    try {
        x.f = 1;
    }
    catch (Exception e) {
        y.f = 10;
    }
}
```

```
atomic {
    try {
        y.f = 1;
    }
    catch (Exception e) {
        x.f = 10;
        z.f = 10;
    }
}
```

(a)

```
boolean coldLocksTaken = false;
lockWrite(x);
try {
    x.f = 1;
}
catch (Exception e) {
    lockWrite(y);
    coldLocksTaken = true;
    y.f = 10;
}
finally {
    unlockWrite(x);
    if (coldLocksTaken)
        unlockWrite(y); }
```

```
boolean coldLocksTaken = false;
lockWrite(y);
try {
    y.f = 1;
}
catch (Exception e) {
    lockWrite(x);
    lockWrite(z);
    coldLocksTaken = true;
    x.f = 10;
    z.f = 10;
}
finally {
    unlockWrite(y);
    if (coldLocksTaken) {
        unlockWrite(x)
        unlockWrite(z); } }
```

(b)

```
boolean coldLocksTaken = false;
lockWrite(x);
try {
    x.f = 1;
}
catch (Exception e) {
    lockWrite(y);
    coldLocksTaken = true;
    y.f = 10;
}
finally {
    unlockWrite(x);
    if (coldLocksTaken)
        unlockWrite(y); }
```

```
boolean coldLocksTaken = false;
lockWrite(y);
try {
    y.f = 1;
}
catch (Exception e) {
    lockWrite(x);
    lockWrite(z);
    coldLocksTaken = true;
    x.f = 10;
    z.f = 10;
}
finally {
    unlockWrite(y);
    if (coldLocksTaken) {
        unlockWrite(x)
        unlockWrite(z); } }
```

(c)

Figure 6.2: Deferring locks can lead to unrecoverable deadlocks, as now locks are not all acquired together at the start of the atomic section. However, an analysis could be performed to identify exactly which deferred locks are the culprits and push them to the start of the atomic section.

instances need to be acquired. Hence, an important area of future work would be to replace type locks with less coarse locks.

6.2.3 Parallelism within atomic sections

Atomic sections are traditionally disallowed from forking threads. However, in the future we might envisage libraries internally using parallelism to perform their computations, especially if we start seeing hundreds of cores in commodity processors. To be able to call such libraries from within atomic sections, means we would need to be able to support threads. This is known as *nested parallelism* [agr08] or *parallel nesting* [bar10] in the space of transactional memory. However, it has not yet been considered for lock inference.

6.2.4 Hybrid with transactional memory

Another interesting area of future work is the combination of transactional memory and lock inference into a single implementation of atomic sections. Transactional memory has very good scalability but is unable to handle irreversible operations well. On the other hand, lock inference has low overhead for when there is lots of contention and is able to handle I/O system calls. Thus, one possible hybrid implementation may use transactional memory by default and then revert to using lock inference when it encounters I/O or if it finds that transactions are rolling back excessively.

6.3 Closing remarks

From the outset, the goal of this work has been to be able to run our analyses on existing real-world Java programs. Given the complexity of these programs, part of the work was to find other tools and techniques to build on. In particular, the IDE analysis framework proved to be a very good foundation upon which to design our analysis, as the pointwise representations of [SRH96, RSX08] afford an efficient implementation. Furthermore, Soot was perfect for

implementing our analyses as it provided the necessary framework and supporting analyses. An implementation would not have been possible without it. Finally, we also wanted to be able to experiment with modifying the run-time, to make it cheap to lookup a lock for an object as well as store and retrieve thread-local data. Jikes RVM provided us with not only a VM that was able to run all our benchmarks but also the ability to experiment in this way. The advantage of having implemented our techniques in such tools is that they can then be used and furthered by others.

Bibliography

[AAK+05] CS Ananian, K. Asanovic, BC Kuszmaul, CE Leiserson, and S. Lie. Unbounded transactional memory. *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 316–327, 2005.

[ABH+09] Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Implementation and Use of Transactional Memory with Dynamic Separation. In Oege de Moor and Michael I Schwartzbach, editors, *CC*, pages 63–77. Springer, 2009.

[ADG+99] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, YS 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.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[agr08] Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174, New York, NY, USA, 2008. ACM.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[AHM09] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In Daniel A Reed and Vivek Sarkar, editors, *PPOPP*, pages 185–196. ACM, 2009.

[AR05] C. Scott Ananian and Martin Rinard. Efficient Object-Based Software Transactions. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, October 2005.

[Art01] C Artho. Finding Faults in Multi-threaded Programs. Master's thesis, ETH Zürich, 2001.

[bar10] Leveraging parallel nesting in transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 91–100, New York, NY, USA, 2010. ACM.

[BCF04] Nick Benton, Luca Cardelli, and Cedric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[BGK+06] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D Berger, and Mark D Corner. Flux: A Language for Programming High-Performance Servers. In *Proceedings of USENIX Annual Technical Conference*, pages 129–142, Boston, MA, May 2006.

[BKMS98] D.F. 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.

[BLM05] C. Blundell, E.C. Lewis, and M.M.K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. *Workshop on Duplicating, Deconstructing, and Debunking (WDDD), June*, 2005.

[Boy04] Chandrasekhar Boyapati. *Safejava: a unified type system for safe programming.* PhD thesis, Massachusetts Institute of Technology, 2004.

[CA04] Bryan Chan and Tarek S Abdelrahman. Run-Time Support for the Automatic Parallelization of Java Programs. *J. Supercomput.*, 28(1):91–117, 2004.

[CCG08] Sigmund Cherem, Trishul M Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In Rajiv Gupta and Saman P Amarasinghe, editors, *PLDI*, pages 304–315. ACM, 2008.

[CDE] Dave Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety.

[CGE08] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep Off the Grass: Locking the Right Path for Atomicity. In Laurie J Hendren, editor, *CC*, pages 276–290. Springer, 2008.

[CGS+99] J.D. Choi, M. Gupta, M. Serrano, V.C. 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.

[CMC+06] Brian D Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle 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.

[com06] Everything2 com. Lock Convoying, February 2006.

[Cun10] David Cunningham. *Locking Atomic Sections*. PhD thesis, May 2010.

[DM05] W. Dietl and P. Muller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.

[DSS06] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. *Proc. International Symposium on Distributed Computing*, 2006.

[EFJM07] Michael Emmi, Jeffrey S Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 291–296. ACM, 2007.

[EGLT76] K.P. Eswaran, J. Gray, R. A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.

[Enn06] Robert Ennals. Software transactional memory should not be obstruction-free. 2006.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.

[FFL05] Cormac Flanagan, Stephen N Freund, and Marina 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.

[FH04] K. Fraser and T. Harris. Concurrent Programming without Locks. *Submitted for publication*, 2004.

[FQ03a] Cormac Flanagan and Shaz 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.

[FQ03b] Cormac Flanagan and Shaz 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.

[FQ03c] S Freund and S Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[FR02] P. Felber and M.K. Reiter. Advanced concurrency control in Java. *Concurrency and Computation: Practice and Experience*, 14(4):261–285, 2002.

[FR04] Mikhail Fomitchev and Eric 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.

[Fra03] Keir Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003.

[GC09] Shu-ling Garver and Bob Crepps. The New Era of Tera-scale Computing, January 2009.

[GHKP05] 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.

[GLP75] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *VLDB '75: Proceedings of the 1st International Conference on Very Large Data Bases*, pages 428–451, New York, NY, USA, 1975. ACM.

[Goe05] Brian Goetz. Synchronization optimizations in Mustang. *Java theory and practice (IBM developerWorks)*, October 2005.

[Gro03] Dan 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.

[Gud07] Khilan Gudka. Implementing Atomic Sections using Lock Inference. Master's thesis, Imperial College London, June 2007.

[Hal08] Richard L Halpert. Static Lock Allocation. Master's thesis, McGill University, April 2008.

[Har03] TL Harris. Design choices for language-based transactions. *University of Cambridge Computer Laboratory Tech. Rep., Aug*, 2003.

[Har05] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[HEM93] 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.

[HF03] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.

[HFP06] Michael Hicks, Jeffrey S Foster, and Polyvios 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.

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

[HG06] 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.

[HLM03] 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.

[HLM06] 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.

[HLMSI03] M. Herlihy, V. Luchangco, M. Moir, and W.N. 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.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, December 2010.

[HMPJH05] 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.

[Hos06] Mike Hoskins. A Java Framework for Building Data-Intensive Applications. Technical report, 2006.

[HP04] David Hovemeyer and William Pugh. Finding Concurrency Bugs in Java. Technical report, 2004.

[HPST06] 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.

[HPV07] Richard L Halpert, Christopher J F Pickett, and Clark Verbrugge. Component-Based Lock Allocation. In *PACT*, pages 353–364. IEEE Computer Society, 2007.

[HRD04] John Hatcliff, Robby, and Matthew B Dwyer. Verifying Atomicity Specifications for Concurrent Object-Oriented Software Using Model-Checking. In *VMCAI*, pages 175–190, 2004.

[HSY04] Danny Hendler, Nir Shavit, and Lena 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.

[Jon97] Mike Jones. What Really Happened on Mars Rover Pathfinder. *ACM Forum on Risks to the Public in Computers and Related Systems*, 19(49), December 1997.

[KB02] David Kalinsky and Michael Barr. Priority Inversion. *Embedded Systems Programming*, pages 55–56, April 2002.

[KCH+06] S. Kumar, M. Chu, C.J. 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.

[KK08] Uday P Khedker and Bageshri Karkare. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In Laurie J Hendren, editor, *CC*, pages 213–228. Springer, 2008.

[kog12] A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 141–150, New York, NY, USA, 2012. ACM.

[KSK09] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC, 2009.

[Lea05] Doug Lea. The java.util.concurrent Synchronizer Framework. *Sci. Comput. Program.*, 58(3):293–309, 2005.

[LH03] O Lhotak and L Hendren. Scaling Java Points-to Analysis Using SPARK. *Lecture Notes in Computer Science*, 2622:153–169, 2003.

[LH08] Ondrej Lhotak and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):3:1–3:53, 2008.

[Lho] Ondrej Lhotak. *Program Analysis Using Binary Decision Diagrams*. PhD thesis.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137, 1977.

[LR07] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan {& Claypool Publishers, 2007.

[LT93] Nancy G Leveson and Clark S Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.

[McC76] T.J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, pages 308–320, 1976.

[MH05] J.E.B. Moss and A.L. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[MHW05] K.E. Moore, M.D. Hill, and D.A. Wood. Thread-level transactional memory. *TR1524, Comp. Science Dept. UW Madison, March*, 31, 2005.

[Moi97] M. Moir. Transparent Support for Wait-Free Transactions. *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

[Mos06] J.E.B. Moss. Open Nested Transactions: Semantics and Support. *Workshop on Memory Performance Issues*, 2006.

[MSS04] V.J. Marathe, W.N. Scherer, and M.L. 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.

[mul07] Ownership transfer in universe types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 461–478, New York, NY, USA, 2007. ACM.

[Myc07] Alan Mycroft. Programming Language Design and Analysis Motivated by Hardware Evolution. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, pages 18–33. Springer, 2007.

[MZGB06] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006.

[NA98] Gleb Naumovich and George 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.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[Ous96] J K Ousterhout. Why Threads Are A Bad Idea (for most purposes). Technical report, January 1996.

[Pea05] David J Pearce. *Some directed graph algorithms and their application to pointer analysis*. PhD thesis, Imperial College of Science, Technology and Medicine, February 2005.

[Pou04] Kevin Poulsen. Tracking the blackout bug. *Security Focus*, April 2004.

[ram03] *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[RD06] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *ACM SIGPLAN Notices*, 2006.

[Rei12] James Reinders. Transactional Synchronization in Haswell, February 2012.

[RG05] 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.

[RHL05] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 494–505, 2005.

[RSX08] Atanas Rountev, Mariana Sharp, and Guoqing Xu. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In Laurie J Hendren, editor, *CC*, pages 53–68. Springer, 2008.

[SATH+06] B. Saha, A.R. Adl-Tabatabai, R.L. Hudson, C.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.

[Sco87] M.L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, 1987.

[sil06] *Operating System Principles*. Wiley-India, 2006.

[SIS05] W.N. Scherer III and M.L. 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.

[SMSAT08] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza 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.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[SRH96] Sagiv, Reps, and Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *TCS: Theoretical Computer Science*, 167, 1996.

[ST95] N. Shavit and D. Touitou. Software transactional memory. *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, March 2005.

[Szy05] Craig Szydlowski. Multithreaded Technology and Multicore Processors. *Dr. Dobb's Journal*, May 2005.

[vos03] OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools, WOMPAT 2003, Toronto, Canada, June 26-27, 2003, Proceedings. In Michael Voss, editor, *WOMPAT*. Springer, 2003.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In Stephen A MacKay and J Howard Johnson, editors, *CASCON*, page 13. IBM, 1999.

[wel08] Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, New York, NY, USA, 2008. ACM.

[WHJ06] A. Welc, A.L. Hosking, and S. Jagannathan. Transparently Reconciling Transactions with Locking for Java Synchronization. *European Conference on Object-Oriented Programming*, 2006.

[Wik06] Wikipedia. Lock convoy. Technical report, December 2006.

[WS06] Liqiang Wang and Scott D Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, 2006.

[ZSZ+08] Yuan Zhang, Vugranam C Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R Gao. Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections. In José Nelson Amaral, editor, *LCPC*, pages 141–155. Springer, 2008.

Appendix A

Output of Halpert et al on Concurrent Hello World program

```
[wjtp.tn] *** Find and Name Transactions *** Wed Jan 19 18:02:53 GMT 2011
[0,0] r0 := @this: ConcurrentPrintln
[0,0] specialinvoke r0.<java.lang.Object: void <init>()>()
[0,0] return
[0,0] r0 := @parameter0: java.lang.String[]
[0,0] $r7 = r0[0]
[0,0] i0 = staticinvoke <java.lang.Integer: int parseInt(java.lang.String)>($r7)
[0,0] $r8 = r0[1]
[0,0] i1 = staticinvoke <java.lang.Integer: int parseInt(java.lang.String)>($r8)
[0,0] r1 = r0[2]
[0,0] $r9 = new java.io.PrintStream
[0,0] $r2 = new java.io.BufferedOutputStream
[0,0] $r3 = new java.io.FileOutputStream
[0,0] $r4 = <java.io.FileDescriptor: java.io.FileDescriptor out>
[0,0] specialinvoke $r3.<java.io.FileOutputStream: void <init>(java.io.FileDescriptor)>($r4)
[0,0] specialinvoke $r2.<java.io.BufferedOutputStream: void <init>(java.io.OutputStream)>($r3)
[0,0] specialinvoke $r9.<java.io.PrintStream: void <init>(java.io.OutputStream,boolean,java.lang.String)>($r2, 1, r1)
[0,0] r5 = $r9
[0,0] r6 = newarray (java.lang.Thread)[i0]
[0,0] i2 = 0
[0,0] goto [?= (branch)]
[0,0] if i2 < i0 goto $r10 = new ConcurrentPrintlnPrintThread
[0,0] $r10 = new ConcurrentPrintlnPrintThread
[0,0] $r11 = new java.lang.StringBuilder
[0,0] specialinvoke $r11.<java.lang.StringBuilder: void <init>(java.lang.String)>("t")
```

```
[0,0] $r12 = virtualinvoke $r11.<java.lang.StringBuilder: java.lang.StringBuilder append(int)>(i2)
[0,0] $r13 = virtualinvoke $r12.<java.lang.StringBuilder: java.lang.String toString()>()
[0,0] specialinvoke $r10.<ConcurrentPrintlnPrintThread: void <init>(java.lang.String,int,java.io.PrintStream)>($r13, i1, r5)
[0,0] r6[i2] = $r10
[0,0] i2 = i2 + 1
[0,0] i3 = 0
[0,0] goto [?= (branch)]
[0,0] if i3 < i0 goto $r14 = r6[i3]
[0,0] $r14 = r6[i3]
[0,0] virtualinvoke $r14.<java.lang.Thread: void start()>()
[0,0] i3 = i3 + 1
[0,0] i4 = 0
[0,0] goto [?= (branch)]
[0,0] if i4 < i0 goto $r15 = r6[i4]
[0,0] $r15 = r6[i4]
[0,0] virtualinvoke $r15.<java.lang.Thread: void join()>()
[0,0] i4 = i4 + 1
[0,0] return
[0,0] r0 := @this: ConcurrentPrintlnPrintThread
[0,0] r1 := @parameter0: java.lang.String
[0,0] i0 := @parameter1: int
[0,0] r2 := @parameter2: java.io.PrintStream
[0,0] specialinvoke r0.<java.lang.Thread: void <init>()>()
[0,0] r0.<ConcurrentPrintlnPrintThread: java.lang.String message> = r1
[0,0] r0.<ConcurrentPrintlnPrintThread: int numPrints> = i0
[0,0] r0.<ConcurrentPrintlnPrintThread: java.io.PrintStream printer> = r2
[0,0] return
[0,0] r0 := @this: ConcurrentPrintlnPrintThread
[0,0] i0 = 0
[0,0] goto [?= $i1 = r0.<ConcurrentPrintlnPrintThread: int numPrints>]
[0,0] $i1 = r0.<ConcurrentPrintlnPrintThread: int numPrints>
[0,0] if i0 < $i1 goto $r2 = new java.lang.Object
[0,0] $r2 = new java.lang.Object
[0,0] specialinvoke $r2.<java.lang.Object: void <init>()>()
prep: r1 = $r2
[0,0] entermonitor $r2
Transaction found in method: <ConcurrentPrintlnPrintThread: void run()>
Warning: using default implementation of addAll. You should implement a faster specialized implementation.
this is of type soot.jimple.spark.sets.HashPointsToSet
other is of type soot.jimple.spark.sets.HybridPointsToSet
exclude is null
[1,0] $r3 = r0.<ConcurrentPrintlnPrintThread: java.io.PrintStream printer>
[1,0] $r4 = r0.<ConcurrentPrintlnPrintThread: java.lang.String message>
{0,0} virtualinvoke $r3.<java.io.PrintStream: void println(java.lang.String)>($r4)
```

```
        Read/Write Set for LibInvoke:
Read Set:(0)[emptyset]


Write Set:(0)[emptyset]


[0,0] exitmonitor r1
[0,0] $r5 := @caughtexception
[0,0] $r5 := @caughtexception
[0,0] exitmonitor r1
[0,0] $r5 := @caughtexception
[0,0] throw $r5
[0,0] goto [?= i0 = i0 + 1]
[0,0] i0 = i0 + 1
[0,0] $i1 = r0.<ConcurrentPrintlnPrintThread: int numPrints>
[0,0] if i0 < $i1 goto $r2 = new java.lang.Object
[0,0] $r2 = new java.lang.Object
[0,0] specialinvoke $r2.<java.lang.Object: void <init>()>()
prep: r1 = $r2
[0,0] entermonitor $r2
[0,0] return
[wjtp.tn] *** Find Transitive Read/Write Sets *** Wed Jan 19 18:02:54 GMT 2011
[wjtp.tn] *** Calculate Locking Groups *** Wed Jan 19 18:02:54 GMT 2011
[wjtp.tn] *** Detect the Possibility of Deadlock *** Wed Jan 19 18:02:54 GMT 2011
[wjtp.tn] *** Calculate Locking Objects *** Wed Jan 19 18:02:54 GMT 2011
[wjtp.tn] *** Print Output and Transform Program *** Wed Jan 19 18:02:54 GMT 2011
```