Keep Off the Grass: Locking the Right Path for Atomicity

Dave Cunningham, Khilan Gudka, and Susan Eisenbach

Imperial College London
{dc04,khilan,sue}@doc.ic.ac.uk

Abstract. Atomicity provides strong guarantees against errors caused by unanticipated thread interactions, but is difficult for programmers to implement with low-level concurrency primitives. With the introduction of multicore processors, the problems are compounded. Atomic sections are a high level language feature that programmers can use to designate the blocks of code that need to be free from unanticipated thread interactions, letting the language implementation handle the low-level details such as deadlock. From a language designer's point of view, the challenge is to implement atomic sections without compromising performance.

We propose an implementation of atomic sections that inserts locks transparently into object-oriented programs. The main advantages of our approach are: (1) We infer *path* expressions (that at run-time resolve to actual objects) for many more accesses in the atomic section than previous work could infer. (2) We use multi-granularity locking for guarding iterative traversals. (3) We ensure freedom from deadlock by rolling back the lock acquisition phase. (4) We release locks as early as possible. In summary, our approach uses a finer-grained locking discipline than previous lock inference techniques.

1 Introduction

In shared memory concurrent software, to prevent erroneous behaviour due to unanticipated thread interactions, programmers ensure that appropriate blocks of code are *atomic* [18]. Atomicity is a stronger property than race-freedom [6], guaranteeing against high-level concurrency bugs such as stale values. A programmer can reason about atomic blocks with sequential intuition.

In an object-oriented language with locks, a block of code can be made atomic through the following process: A guarding discipline must be chosen (and obeyed) that specifies a lock for each shared memory object. For each block of atomic code and for each object accessed by the block, the appropriate locks must be acquired. Thus the developer must be aware of accesses internal to any invoked functions (breaking encapsulation). Furthermore, for correctness, the lock acquisitions and releases follow a two-phase discipline [5], i.e. the acquisitions must precede all releases in the block. Finally, locks must be acquired according to the same partial order to avoid deadlocks. In time, it will become necessary to maintain the code, where care must be taken to keep the lock acquisitions in sync with

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 276–290, 2008.

newly introduced accesses in the code. The process is unforgiving. Any error introduces a possible bug (e.g. deadlock, stale value, race condition), which is hard to reproduce, and troublesome to trace to an underlying cause.

While atomicity allows us to more confidently assert the absence of concurrency errors, it cannot be reliably enforced by programmers. The problem of enforcement would be better solved by the programming language implementation, and the primitive that sets out to achieve this goal is the *atomic section*. By simply marking a block of code as atomic, the programmer can be assured that the implementation will execute as if all other threads were suspended, without having to implement any extra machinery.

Although atomic sections allow the programmer to pretend a block is executed sequentially, such an implementation would have poor performance. A transparent optimisation is for non-interfering threads to be allowed to execute in parallel with atomic sections. There are a number of proposed methods for efficient atomic section implementations, which can be divided into two categories: Optimistic approaches, in the form of Software or Hardware Transactional Memory [2,11,12,14,17,20,21,22,23], rely on being able to detect thread interference at runtime and rollback the state to the beginning of the atomic section where it is known to be uncorrupted. Pessimistic approaches statically attempt to infer locks sufficient for preventing interference [4,7,15,19,25]. The more efficient implementations of transactions and all lock inference implementations, including ours, prohibit the access of shared objects outside of atomic sections.

The optimistic approach can detect interference, whereas the nature of static inference means that only a conservative approximation can be made, reducing parallelism by taking more locks than required for a particular execution of an atomic section. On the other hand, the pessimistic approach does not require any runtime machinery for recording accesses, which if implemented in software can reduce performance. Additionally, no cycles will be wasted contributing towards a state which gets rolled back. If contention is high, transactional systems may spend more time rolling back than making useful progress. Finally, only internal invisible actions such as memory reads and writes can be rolled back; if an optimistic approach is to be used, external actions such as IO must not be allowed in an atomic section. This means that the compiler either has to reject such programs or transparently move the IO out of the atomic section. Pessimistic techniques have no such restriction.

We chose to implement atomic sections pessimistically using lock inference. We face the same challenges as programmers: Taking enough locks at the right time, striving for fine-grained locking while also trying to minimise the overhead of locking code, and avoiding deadlocks. We have strived to make our implementation transparent to the programmer, requiring no additional type or other annotations.

In Sect. 2, we summarise our approach and describe how we handle an example. In Sect. 3, we describe our program analysis in detail. In Sect. 4, we describe how we use the result of our program analysis to generate locking code, without deadlocking. In Sect. 5, we report on the use of our approach with part of a real

program. In Sect. 6, we discuss how our approach differs to that of the related work. We conclude with Sect. 7.

2 General Approach and Features

We use a data-flow analysis, at link or JIT time, to infer the object accesses performed by each atomic section. This analysis needs to traverse any code that might be invoked by the block in question, so the whole program is needed. When the analysis terminates, we know, at each program point, the set of objects that are accessed from that point until the end of the atomic section. The inferred accesses then need to be translated into locks. We believe our representation of accesses is novel, and the most precise to date. As a simplification all objects after construction are shared. Detecting thread-local heap objects would benefit many systems like ours, and we do not discuss it here.

We try and use one lock per object, or *instance* locks, where possible, so that the parallelism can scale with the data. Sometimes code can access a statically unbounded number of objects. This happens during iterations over objects, and when we approximate an array index expression. In such cases, we use the type of the accessed objects to take a *multilock* which guards all instances of that type and subtypes. The semantics of multilocks require that if one thread has taken the multilock on an object, any other threads attempting to lock a subordinate instance of that multilock will be blocked until the multilock is released. We also distinguish between read/write accesses, and we use read/write locks to allow multiple parallel reads. We need re-entrant locks in case objects happen to be aliased at runtime, causing the same lock to be taken twice.

The code in Figure 1 is for an instant messaging system, where a client can send a stream of messages to another client, by name, through a central server. The `Client` constructor registers a new client in a centralised hashtable of clients. This must be an atomic operation in order that the uninitialised value of the hash entry `e.val` is not visible to other threads. Our system infers two locks – the hashtable itself (for reading), and the array of buckets inside the hashtable (for writing). Although the hash entry is modified, it is a newly constructed object and thus cannot be seen by other threads. We give the inferred locks as comments in the code, where ! denotes a write lock.

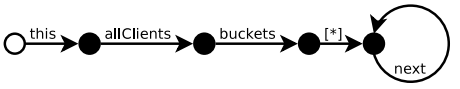
The atomic section starting on line `(53)` iterates (`HashEntry.findKey` is recursive) through a list of hash entries, and thus the analysis has to lock (for reading) the multilock that subsumes every hash entry. Atomic sections starting on lines `(59)` and `(63)` simply access a pair of clients and a single client object, and the locking reflects this. The atomic section starting on line `(45)` is only ever called from within another atomic section starting on line `(59)`, so does not have any locking code inserted into it. If it were also called from a *pre-emptive* context (i.e. from outside an atomic section), we would have a problem inserting locking code into it, because this code would also be executed by the atomic section starting on line `(59)`. We solve this problem by duplicating functions if they are called from both atomic and pre-emptive contexts.

```
1  class HashEntry {
2      string key; Object val; HashEntry next;
3      HashEntry (string key) { this.key = key; }
4      HashEntry findKey (string key) {
5          if (this.key==key) {
6              return this;
7          } else {
8              if (next==null) { return null; }
9              else { return next.findKey(key); }
10 }   }   }
11
12 class HashTable {
13     HashEntry[] buckets;
14     HashTable() { buckets = new HashEntry[100]; }
15     int index(string key) {
16         int hash = key.hash % buckets.length;
17         if (hash<0) { hash = hash + buckets.length; }
18         return hash;
19     }
20     HashEntry createHashEntry(string key) {
21         HashEntry entry = new HashEntry(key);
22         int index = index(key);
23         entry.next = buckets[index];
24         buckets[index] = entry;
25         return entry;
26     }
27     HashEntry findHashEntry(string key) {
28         HashEntry entry = buckets[index(key)];
29         if (entry==null) { return null; }
30         return entry.findKey(key);
31 }   }
32
33 class Client {
34     string name;  HashTable allClients;  Client interlocutor;
35     Client (HashTable allClients, string name) {
36         this.allClients = allClients; this.name = name;
37         atomic {  //locks: {allClients, !allClients.buckets}
38             HashEntry e = allClients.createHashEntry(name);
39             e.val = this;
40         }
41         run();
42     }
43     string read() { return ""; }
44     void accept(Client source, string msg) {
45         atomic {  //omitted
46             print "<"+source.name+"> ---> <"+name+"> "+msg;
47     }   }
48     void run() {
49         while (true) {
50             string msg = read();
51             if (msg=="connect") {
52                 string name = read();
53                 atomic {  //locks: {HashEntry}
54                     HashEntry e = allClients.findHashEntry(name);
55                     interlocutor = (Client) e.val;
56             }   }
57             if (msg=="send") {
58                 string cargo = read();
59                 atomic {  //locks: {this, interlocutor}
60                     interlocutor.accept(this,cargo);
61             }   }
62             if (msg=="disconnect") {
63                 atomic {  //locks: {!this}
64                     interlocutor = null;
65 }   }   }   }   }
```

Fig. 1. Example source program using atomic sections

Table 1. Example of path graphs inferred at the top of two atomic sections from Fig. 1

Line	Path set	Path graph
37	`allClients` `allClients.buckets`	

53	`this`
	`this.allClients`
	`this.allClients.buckets`
	`this.allClients.buckets[*]`
	`this.allClients.buckets[*].next`
	`this.allClients.buckets[*].next.next`
	`...`
		

Our program analysis gives us information about what locks should be held at every program point in the atomic section. This means we have enough information to release locks straight after the last access of any objects they guard. Releasing locks early reduces contention, at no extra cost.

3 Path Graphs Inference

The keystone of our approach is an analysis that infers the object accesses in a given block of code. We assume that a control flow graph (CFG) is set up, with five types of node: **Copies** e.g. `x=y` (including assignment of `new` objects or `null`), **stores** e.g. `x.f=y`, **loads** e.g. `x=y.f`, **array stores** e.g. `x[i]=y`, and **array loads** e.g. `x=y[i]`. Our analysis is a backwards 'may' analysis. Each edge initially has no accesses, and we don't introduce accesses at the exit of the atomic section. Instead, accesses are *generated* by the CFG nodes (except for copy nodes), which also *transform* accesses. When the analysis terminates, the complete set of object accesses for the atomic section is left at its entry node.

The state at each edge is a *path graph*, which we use to represent a possibly-infinite set of *paths*. A path is a sequence of field or array accesses starting from a local variable, and can be used to statically characterise an object access. When statically analysing an iteration over an object structure, we do not know how many times the loop will repeat, and thus how many objects will be touched. Although the analysis can infer an infinite number of paths, the path graph representation is finite. Table 1 gives an example of two path graphs as produced by our analysis, and their corresponding path sets. The path sets are prefix-complete since we cannot access an object unless it is either bound to a variable before the atomic section began, or can be retrieved through another object.

In general, it will be impractical to record which element of an array was accessed, e.g. if the index was calculated using a complicated algorithm. The syntax `[*]` in the paths represents this. For brevity, in this presentation we immediately widen array element accesses to `[*]`. A system which already has some understanding of integer arithmetic could be more precise.

Path graphs are deterministic finite automata (DFA). The hollow node is the initial state, and all the other nodes are possible exit states, thus the set of paths is represented by the path graph. The atomic section starting on line `(53)`, which

$$
\begin{aligned}
a[x = y]^n &= \emptyset \\
a[x = \texttt{null}]^n &= \emptyset \\
a[x = \texttt{new}]^n &= \emptyset \\
a[x = y.f]^n &= \{y \to n\} \\
a[x.f = y]^n &= \{x \to n\} \\
a[x = y[i]]^n &= \{y \to n\} \\
a[x[i] = y]^n &= \{x \to n\}
\end{aligned}
$$

$$
\begin{aligned}
t[x = y]^n(G) &= G \setminus \{x \to n' | x \to n' \in G\} \cup \{y \to n' | x \to n' \in G\} \\
t[x = \texttt{null}]^n(G) &= G \setminus \{x \to n' | x \to n' \in G\} \\
t[x = \texttt{new}]^n(G) &= G \setminus \{x \to n' | x \to n' \in G\} \\
t[x = y.f]^n(G) &= G \setminus \{x \to n' | x \to n' \in G\} \cup \{n \xrightarrow{f} n' | x \to n' \in G\} \\
t[x.f = y]^n(G) &= G \setminus \{n' \xrightarrow{f} \_ | x \to n' \in G, (\nexists z \neq x : z \to n' \in G), (\nexists n''' : n''' \rightharpoonup n' \in G)\} \\
&\quad \cup \{y \to n' | \_ \xrightarrow{f} n' \in G\} \\
t[x = y[\_]]^n(G) &= G \setminus \{x \to n' | x \to n' \in G\} \cup \{n \xrightarrow{[*]} n' | x \to n' \in G\} \\
t[x[\_] = y]^n(G) &= G \cup \{y \to n' | \_ \xrightarrow{[*]} n' \in G\}
\end{aligned}
$$

Fig. 2. Data flow transfer functions

through iteration can access an unbounded number of objects, requires a cycle in the path graph in order to keep the representation finite. We represent the path graph as a set of edges, which are either labelled between a pair of nodes, or unlabelled between a variable and a node, e.g. the second path graph in Table 1 could be represented with the set:

$$\{this \to 1, 1 \xrightarrow{allClients} 2, 2 \xrightarrow{buckets} 3, 3 \xrightarrow{[*]} 4, 4 \xrightarrow{next} 4\}$$

As this representation allows multiple identically-named arrows from a state, it is actually a nondeterministic finite automaton (NFA). Our analysis computes NFAs but for locking purposes we convert them to minimised DFAs [16]. The numbered nodes in this representation correspond to the index of the CFG node where the access occurred. Because there are finitely many CFG nodes, variables, and fields, the set of edges and therefore the state of the analysis is also finite. It remains to give the transfer functions that compute, at each CFG node, the entry set from its exit set.

Figure 2 gives two functions, where $(\setminus)$ binds tighter than $(\cup)$, x, y, z range over variables, f, g range over fields, G ranges over path graphs, and _ represents an unbound variable. Function a gives a path graph representing accesses generated by the given CFG node with index n. This path graph has meaning only in the state where the access occurred. Since we intend to lock the objects represented by this path graph at the top of the atomic section, and the state is mutated by assignments between these two points, the function t will transform a path graph to compensate for the side-effect of a given CFG node. At each CFG node, we compute the entry path graph by applying t to the exit path graph, and also include the result of a. Translating allows us to handle code like `atomic {x=y ; x.f=42}` without requiring that `x` and `y` are guarded by the same lock. Our analysis returns the path `y`.

When translating path graphs through copy nodes, paths starting with the lvalue are renamed so they start with the rvalue, except where the rvalue is a `new` object or `null`, where the accesses are simply killed. Load nodes are similar, but replace accesses of the form `x.f1.f2` with `y.f.f1.f2`. Note that this works only

because a sets up an edge from `y` to n. Store nodes have to handle aliases, e.g. in `atomic { me.car = you.car ; dave.car.fuel = 100 }` it is clear that `me`, `you`, and `dave` are accessed, but it is not clear which car is accessed. If `me` was an alias of `dave` then the accessed car is described by `you.car`, otherwise the assignment has no effect and we must lock `dave.car`. Alias analysis can help here, but in this presentation we assume conservatively that everything can be an alias, hence the function t introduces new edges from `y` to any node that has the appropriate field edge leading to it. The only alias we can assert is that `x` is an alias of `x`; in other words, we can kill `x.f` from the graph. This means killing an edge like $\xrightarrow{f}$, which we can do only if it is not used by any other paths, which we require with the two negated existential predicates. Array loads and stores are similar to their field counterparts.

It is possible to improve the accuracy of the analysis using type information, e.g. if `x` : `A[]` and `z` : `B[]` in `atomic { x[1] = y ; z[1].f = 10; }` then there is no need to infer the path `y`. Since the two arrays have different types, `x` cannot be an alias of `z` and thus the access `z[*]` will suffice. Similarly, we can use type information to distinguish between identically-named fields in different classes. Points-to information could also be used.

4 Lock Insertion

In this section we describe in detail our approach for inserting lock acquisition and release code into an atomic section. We describe how we detect deadlock and roll back the locking code, and how the analysis supports readers/writers and early unlocking.

4.1 Inferring Locks from a Path Graph

The path graphs analysis outputs a minimised DFA. We first process this graph looking for cycles and widened array access edges (i.e. $\xrightarrow{[*]}$ edges). Nodes reachable from such edges are marked as dirty, and the edges are removed. The graph is now acyclic and we perform a depth first search to pick out paths to lock. In the atomic section on line `(53)` of Table 1 we dirty the far-right node and remove the two edges pointing to it. We infer the paths {`this`, `this.allClients`, `this.allClients.buckets`}, that can be locked directly (in prefix order); we have to use a multilock to lock the dirty node. We use type information (the element type of the `buckets` array) to get the type of the objects represented by the dirty node (`HashEntry`), and we lock the multilock associated with this type. We associate a multilock with each class that guards all instances of the class. This lock allows subordinate instance locks to be acquired only when it is not held. Because classes are subsumed by subtyping, we also lock any subtypes of the class, of which there are none in our example. We also have multilocks for array types, e.g. for `Object[x][y]` accesses.

We wanted to use read/write locks, i.e. locks that allow multiple threads to have the read lock so long as no thread has the write lock. We were surprised

to discover that the path graph representation already encodes this information. Because each node in the graph is a specific CFG node, and the CFG node determines the access type (load/store = read/write), the node index can be used to determine read/write locking. The only caveat is that we have to make sure that we preserve this information when collapsing the NFA to a minimised DFA.

Consider `atomic { y = x.f ; y.g = 10 }`, for which we would lock `x, !x.f`. If `x` was `null`, then this block would throw a `NullPointerException` (NPE). More importantly, so would our locking code. In general, throwing an exception from the locking code would not preserve the semantics of the atomic section, as there may be side-effects from before the NPE that now never occur. We need to either check for `null` before locking, or catch the NPE. If `x` is `null` we do not lock it, or `x.f`. There is a similar problem with `ClassCastException`, e.g. `atomic { y = ((A)x).f ; y.g = 10 }`, for which we also infer `x,x.f`. If the type of `x` does not have a field `f`, then we must cast it to `A`. We must therefore either check that that this is possible, or catch the exception. Null pointer analysis and points-to information can minimise the number of checks required.

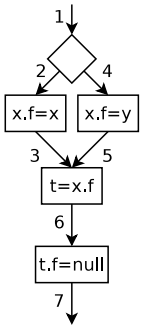
4.2 Deadlock

Existing approaches guarantee the absence of deadlock at compile-time by always acquiring locks in the same order, and if such an ordering cannot be found they typically coarsen the granularity. Our type multilocks are static, thus can be statically ordered, e.g. alphabetically. Our instance locks cannot be statically ordered, but rather than coarsen the granularity, we detect deadlock and roll back the lock acquisition phase at run-time. Although this sounds like a transaction, there are no side effects to roll back, so no transaction log is required. This means that we have to take all locks at the beginning of the atomic section. We expect deadlock to be rarer than transaction collision, because the lock acquisition phase is much shorter than a whole atomic section, so we do not anticipate live locking to result. It also offers the possibility of high-priority threads forcing low-priority threads to give up their locks if they are stalled in a lock-acquisition phase.

Deadlock is typically detected at run-time by looking for cycles in a waits-for graph. Many systems check for deadlock, such as the Java Hotspot VM. This information is available at runtime, as it is needed by the lock implementation, but checking for cycles at each failed lock acquisition still incurs a performance penalty. This is a trade-off however, as we have better granularity as a result of keeping instance locking. If there are more CPUs than awake threads, this computation will be performed on a CPU which would otherwise be idle, eliminating the runtime penalty.

4.3 Parole

If an atomic section has finished accessing an object, but still has a lot of computation left to complete, it can release the lock early to gain finer granularity. We can also demote write locks to read locks, and multilocks to instance locks.



Edge	Fixed point	Early unlocks	Late relocks
1	$\{!x, !y\}$	$\{\}$	$\{\}$
2	$\{!x\}$	$\{!y\}$	$\{\}$
3	$\{x, !x.f\}$	$\{\sout{!x}\}$	$\{x,\sout{!x.f}\}$
4	$\{!x, !y\}$	$\{\}$	$\{\}$
5	$\{x, !x.f\}$	$\{!x,\sout{!y}\}$	$\{x,\sout{!x.f}\}$
6	$\{!t\}$	$\{x,\sout{!x.f}\}$	$\{\sout{!t}\}$
7	$\{\}$	$\{!t\}$	$\{\}$

Fig. 3. Result of path graph analysis applied to early release

This is particularly important in the context of conservative analysis. Figure 3 is a simple example that demonstrates much of the power of early unlocking. We provide, at each edge, the path graph at the fixed point of the analysis, after it has been converted to a DFA, minimised, and reduced to locks. An exclamation mark represents a write lock. All of the locks in this example are paths.

At the entry of the atomic section (edge 1), we take the locks $\{!x, !y\}$ using the deadlock detection/rollback strategy described above. Once this is successful, execution will follow one of the branches. One branch (4,5) will cause us to eventually access both objects, and the other (2,3), only $!x$, as reflected by the edges 2 and 4 in the table. The analysis records that at edge 2, only the lock $!x$ is required, so by subtracting the two sets we can calculate locks to release, i.e. the *Early unlocks* column. If the set of locks increases, e.g. from edge 2 to edge 3, we have to acquire the extra lock in order for a later release to be well-balanced, we list this in the *Late relocks* column. Such locks have already been acquired by the thread, reacquiring them just increases the re-entrant counter. There is no risk of deadlock. We always acquire before release, to make sure we do not let the re-entrant counter reach zero. Thus locking remains two-phase.

At an assignment, where locks are translated from one variable to another, there will usually be a pair of redundant acquire/release actions. We used a simple alias analysis to remove redundant locking, denoted with strike-through. This also allows us to avoid redundant locking on known-to-be null or newly constructed objects. The remaining acquires and releases serve useful purposes: Unlocking $!y$ at edge 2 allows other threads to proceed in parallel, as this thread is now certain it will not write to (or read) y. Releasing x and $!t$ at edges 6 and 7 respectively allows the atomic section to terminate with all locks released. Acquiring x and releasing $!x$ at edge 5 allows other threads to read in parallel, as this thread no longer needs to write to x. The analysis similarly demotes a multilock to an instance lock, e.g. when a list is searched for an object which is then accessed. In the case of array indexing using a computed index, even if the analysis was equipped with more powerful array access edges and transfer functions, we would have to take a multilock as we would not statically know what element will be accessed. However, once the array index has been computed,

the analysis would demote the multilock to an instance lock and similarly allow other threads to proceed.

The spurious lock of x at edge 3 is necessary because at this time x and $x.f$ are aliases, and the releases at edges 6 and 7 will serve to release the same lock twice. Therefore, in the branch we must acquire x once more to balance the forthcoming releases. Aside from using the alias analysis to remove redundant locking, we were surprised that we needed no extra mechanism to set up early lock release. Our path graph analysis turned out to be powerful enough to encode read/write information and early release information in its basic form.

Sowing unlocking code throughout the invoked functions of an atomic section causes problems when one of the functions is called from more than one context. A number of solutions present themselves: Aggressively in-lining functions is simple and minimises contention, but will increase the size of the program and may cause performance problems such as cache misses. It should be possible to release locks after the call has returned, but a reference would have to be kept to the objects in question in case re-assignment renders them inaccessible. Alternatively, we could acquire a given lock once for each access, and release it after each access, but this would introduce more overhead. We chose to use the first technique, duplicating methods to ensure they are only called from more than one context in the case of recursion.

4.4 Splitting the Atom

Sometimes, it is desirable to turn off the atomicity of an atomic section for a period, perhaps to do some lengthy thread-local computation, or to communicate with other threads. It is possible to refactor the code into two separate atomic sections, but challenging because atomic sections are lexically scoped. To allow easy expression, we use a `preempt` section which when placed inside an atomic section, releases/reacquires locks to break the atomic section into two distinct parts. We have used this construct to implement wait/notify semantics.

The implementation of preempt sections uses the same program analysis as the implementation of atomic sections. A preempt section is represented in an atomic section's CFG by a *black hole* node that blocks the propagation of the path graph. This induces lock releases before the preempt, and acquisitions after it. We have to detect deadlock and backtrack when re-acquiring the locks after the preempt section. Figure 4 is the CFG of an atomic section that called `wait()` between a pair of object accesses. We have annotated the edges with the fixed point locks, acquires, and releases, as in Fig. 3. The grey circular node is the black hole node. The code for the wait/notify implementation is in Fig. 5.

5 Experiment

We have attempted to evaluate our approach on an existing application. We have implemented a subset of Java, with primitive types, classes, reference types, arrays, inheritance, overloading, dynamic binding, branches, loops, and early returns. All the examples in the paper were written in this language.

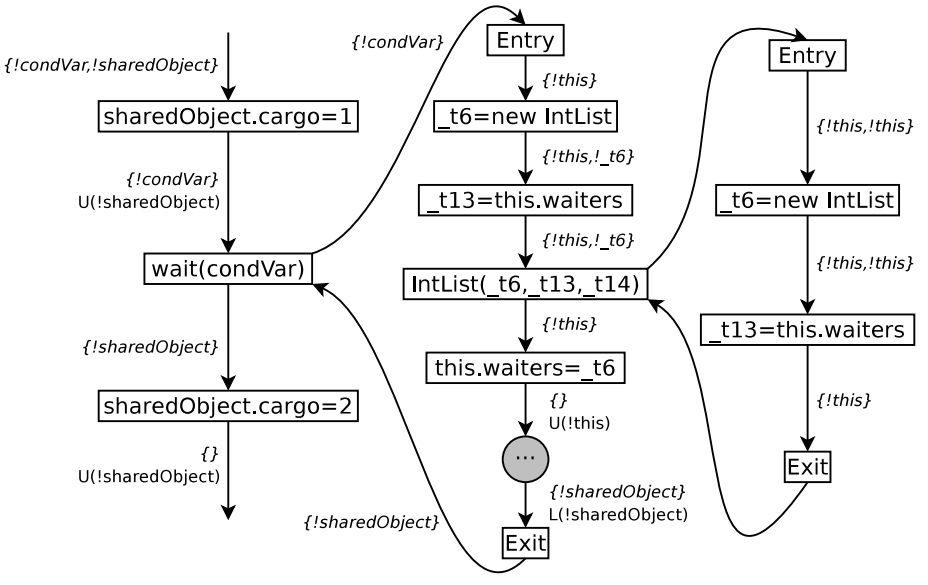


Fig. 4. CFG of an atomic section that calls `wait()`. The fixed point of the analysis is in italics, early lock release is in roman, e.g. U(!this). We have omitted redundant lock acquires and releases. The two `this` locks in the *far right* function correspond to the `this` variables in the `wait()` function and the converter, which are distinct. Our `sharedObject` is an instance of `IntList`, given in Fig. 5.

```
class IntList {
        IntList next; int cargo;
        IntList (IntList next, int cargo) {
                this.next = next;
                this.cargo = cargo;
}       }

class ConditionVariable {
        IntList waiters;
        void wait() {
                atomic {
                        waiters = new IntList(waiters,threadid);
                        preempt { park; }
        }       }
        void notify() {
                atomic {
                        unpark waiters.cargo;
                        waiters = waiters.next;
        }       }
        void notifyAll() {
                atomic {
                        while (waiters!=null) { notify(); }
}       }       }
```

Fig. 5. The code for the wait/notify implementation referenced above. The `IntList` records a queue of waiting threads, so that `notify` can wake them up. The keyword `threadid` gets the id of the current thread. The `unpark` and `park` keywords allow suspending and resuming of threads, by thread id.

Previous work [19,4] has used the AOLserver [1] source code (which is publicly available) as a case study. AOLserver is a high performance http server, written in C, which uses Tcl as a scripting language to drive dynamic content. Although multiple Tcl interpreters can be running simultaneously, e.g. for simultaneous client connections, they are essentially orthogonal. The actual concurrency is handled with C code where a shared store is provided for communication between Tcl interpreters. The source code was annotated with atomic sections and a form of guard annotation (which we do not need) by the authors of [19], who kindly made the source code available to us. The authors of [4] have a lock inference algorithm which they also benchmarked with the annotated AOLserver. Their approach was very different to ours, and we were interested in comparing their results with ours.

Since our analysis has not been implemented for C, we would have to translate AOLserver in order to process it. AOLserver is a large piece of software, so we chose one particular compilation unit, `tclvar.c` which the previous work seemed to find most demanding. We tried to reproduce the original code as closely as possible. Although it was written in C, the code slipped quite easily into an object-oriented design. Because our analysis is at this stage a whole-program analysis, we also had to implement a small part of the Tcl API. Some of the Tcl functions we left as stubs or only partially implemented, but we were careful to reproduce any accesses the real Tcl API would perform. As such, we are actually analysing more code than the previous work.

Although our implementation was naive, using Java data structures to store the analysis state, we hoped to get reasonable times for the AOLserver code. For each atomic section, we recorded how many nodes were present in the CFG, and how long it took to solve on a P4 3.2GHz CPU with 1GB of RAM, running the Java Hotspot VM v1.6.0. We also give the number of read/write multilocks (RM,WM), and how many read/write instance locks (RI,WI) taken at the top each atomic section. These results are presented in Fig. 6. The solve time includes the time spent minimising the path graph and inferring a set of locks at each edge. Our analysis is much faster than [4], as it works directly on the CFG without a costly setup phase, and the solving phase is faster too. Since each atomic section is treated separately, the time is linear in the number of atomic sections. Most of the time is spent cloning and garbage collecting.

6 Related Work

A number of techniques exist to verify programmer implementations of atomicity, such as: type checking [3], type inference [8], model checking [13], theorem proving [9] and run-time analysis [26]. Some of this work has used variants of ownership types, to specify the guarding relationship between objects. Consequently, they can support more powerful guarding disciplines, where static locks can be avoided altogether, but at the cost of additional annotations.

As discussed in Sect. 2, transactional memory is an optimistic implementation of atomicity, but relies on detecting interference from other threads and the

Atomic section	CFG Node count	Solve time (seconds)	RI	WI	RM	WM
1	219	0.566	2	4	4	0
2	242	0.489	2	4	4	0
3	284	0.728	1	4	3	1
4	203	0.345	3	3	3	0
5	319	0.946	1	4	3	1
6	277	0.694	2	4	5	0
7	101	0.119	1	0	4	1
8	286	0.897	2	4	4	0
9	360	2.385	2	4	7	0
10	169	0.164	3	3	4	0
11	272	0.759	1	4	5	0

The results in [4] state 2399 seconds for this file. Our total time was approximately 8 seconds.

Fig. 6. Results of applying our analysis to the AOLserver `tclvar.c` fragment

ability to roll back. It can allow more parallelism but at the cost of IO, and the risk of livelock. Software transactional memory has significant runtime overhead.

More recently, there have been several proposals for lock inference. Initially, [7] was an extension of the authors' type checking techniques, where incomplete synchronisation could be statically corrected. The authors admit that the added locks could introduce deadlock, whereas the following proposals all guarantee absence of deadlock through establishing a static order on lock allocations. A from-scratch proposal [19] required guard annotations, which although allowed the expression of instance locks as well as static locks, seemed not to be as powerful as the object-object relationships in e.g. [8]. Additionally, programs were rejected if the assignments interfered with the guarding discipline (they did not translate paths as we do).

The approaches in [25,15] did not require guard annotations, using just points-to information to indicate static locks. They handle assignment by increasing the granularity so that the lock guarding the lvalue and rvalue are the same. While [25] did have annotations, they were for the orthogonal purpose of partitioning an object's fields into separately-locked groups, as opposed to the typical approach (which we follow) where all an objects fields are guarded by the same lock. Allowing more parallelism, recent work [4,10] has allowed instance locks without needing annotations, but only in special aliasing circumstances.

7 Conclusion and Future Work

By solving the deadlock problem with a runtime technique, we were able to take instance locks rather than static locks. Instance locking is an opportunity for finer granularity and more parallelism. We embrace this opportunity by statically inferring accesses (in the form of paths), without having to reject assignments to variables containing accessed objects. Unlike previous work, we do not merge the locks of different objects if they once resided in the same variable.

In order to improve the speed of the analysis, it would be useful to analyse a method in isolation. Then this method summary (consisting of new accesses and translations) could instantiated in the various places the method is called.

It may be necessary, when dealing with e.g. native code that can't be analysed, to provide such a summary manually, to act as a realistic stub.

Using thread-local-object and may-happen-in-parallel analyses such as those in [10], it would be possible to omit some of the lock acquisitions. This would reduce the overhead, and less time spent acquiring locks would mean fewer deadlocks resolved at runtime. Also, if a group of locks are always acquired together, or not at all, it makes sense to join them into a single lock, also reducing the overhead of lock acquisition.

If it were possible to infer some form of ownership annotations [3], we might be able to use instance locks instead of static locks to handle iterations over objects, and non-trivial array indexes. We would like to try using points-to information, both as a foundation for multilocks (instead of using type information as we currently do), to see if this results in better granularity. Points-to information could also be used by the path graph analysis, to avoid adding so many edges at store CFG nodes.

Our aim was to create a finer-grained locking discipline than that used by previous lock inference techniques. This paper describes our approach which produced promising results. We are now moving our analysis to full Java, using the Soot [24] framework.

Acknowledgements. We are grateful to EPSRC and Microsoft for supporting this work. We also thank Tim Harris and the Slurp research group at Imperial College for interesting discussions, Tristan Allwood for proofreading, and Dossy Shiobara and the AOLserver project for helpful correspondence.

References

1. AOLserver, a highly scalable, multi-threaded application server, `http://aolserver.com/`
2. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium, pp. 316–327 (2005)
3. Cunningham, D., Drossopoulou, S., Eisenbach, S.: Universe Types for Race Safety. In: VAMP 2007, pp. 20–51 (September 2007), `http://pubs.doc.ic.ac.uk/universes-races/`
4. Emmi, M., Fischer, J.S., Jhala, R., Majumdar, R.: Lock allocation. In: POPL 2007: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 291–296. ACM Press, New York (2007)
5. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM 19(11), 624–633 (1976)
6. Flanagan, C., Abadi, M.: Types for safe locking, 91–108 (1999)
7. Flanagan, C., Freund, S.N.: Automatic Synchronization Correction. In: Synchronization and Concurrency in Object-Oriented Languages (SCOOL) (2005)
8. Flanagan, C., Freund, S.N., Lifshin, M.: Type inference for atomicity. In: TLDI 2005: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation, pp. 47–58. ACM Press, New York (2005)

9. Freund, S., Qadeer, S.: Checking concise specifications for multithreaded software. In: Workshop on Formal Techniques for Java-like Programs (2003)
10. Halpert, R.L., Pickett, C.J.F., Verbrugge, C.: Component-based lock allocation. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, Springer, Heidelberg (2007)
11. Harris, T., Fraser, K.: Language support for lightweight transactions. ACM SIGPLAN Notices 38(11), 388–402 (2003)
12. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 48–60 (2005)
13. Hatcliff, J., Robby, D.M.B.: Verifying atomicity specifications for concurrent object-oriented software using model-checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 175–190. Springer, Heidelberg (2004)
14. Herlihy, M., Eliot, J., Moss, B.: Transactional Memory: Architectural Support For Lock-free Data Structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289–300 (1993)
15. Hicks, M., Foster, J.S., Pratikakis, P.: 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)
16. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
17. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid transactional memory. In: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 209–220 (2006)
18. Lomet, D.: Process structuring, synchronization, and recovery using atomic actions. ACM SIGOPS Operating Systems Review 11(2), 128–137 (1977)
19. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. ACM SIGPLAN Notices 41(1), 346–358 (2006)
20. Moore, K.E., Hill, M.D., Wood, D.A.: Thread-level transactional memory. TR1524, Comp. Science Dept. UW Madison (March 31, 2005)
21. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: Proceedings of the 32nd International Symposium on Computer Architecture, pp. 494–505 (2005)
22. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing, pp. 240–248 (2005)
23. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, pp. 204–213 (1995)
24. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: CASCON 1999: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, p. 13. IBM Press (1999)
25. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. SIGPLAN Not. 41(1), 334–345 (2006)
26. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Softw. Eng. 32(2), 93–110 (2006)