Lock Inference in the Presence of Large Libraries

Khilan Gudka[1], Tim Harris[2], and Susan Eisenbach[1]

[1]Imperial College London
{khilan,susan}@imperial.ac.uk
[2]Microsoft Research Cambridge
tharris@microsoft.com

Abstract. Atomic sections can be implemented using lock inference. For lock inference to be practically useful, it is crucial that large libraries be analysed. However, libraries are challenging for static analysis, due to their cyclomatic complexity.
Existing approaches either ignore libraries, require library implementers to annotate which locks to take or only consider accesses performed upto one level deep in library call chains. Thus, some library accesses may go unprotected, leading to atomicity violations that atomic sections are supposed to eliminate.
We present a lock inference approach for Java that analyses library methods in full. We achieve this by (i) formulating lock inference as an Interprocedural Distributive Environment dataflow problem, (ii) using a graph representation for summary information and (iii) applying a number of optimisations to our implementation to reduce space-time requirements and locks inferred. We demonstrate the scalability of our approach by analysing the entire GNU Classpath library comprising 122KLOC.

1 Introduction

Atomic sections [1] are an abstraction for shared-memory concurrency. They allow a programmer to demarcate a block of code that should execute without interference from concurrent threads but leave the low-level details of achieving this to the compiler and/or run-time. If used correctly, they can remove many of the problems that have plagued programmers for decades, such as low-level race conditions, deadlock, priority inversion and convoying [2]. With current abstractions, the frequency of such unfortunate encounters is only likely to increase, given that multi-core processors are the norm [3, 4].

Atomic sections are a language-level construct, hence an important question is, *how should we implement them?* Software Transactional Memory (STM) [5] is a popular approach, wherein memory updates are buffered during execution and then committed atomically. If a conflicting update has already been committed by another thread, the buffer is discarded and the transaction is re-executed (*rollback*). This ability to abort and re-run is essential to STM, as implementations are typically *optimistic*; they execute with the assumption that interference is unlikely to occur. Rolling back execution is unappealing because irreversible

operations (e.g. system calls) cannot be rolled back and performance can be harmed by maintaining undo-logs to allow rollback.

In light of these shortcomings, pessimistic alternatives have been proposed, based on *lock inference*. These alternatives statically infer enough locks to prevent interference and instrument the program with the corresponding lock operations. Since lock inference must consider all possible execution paths, this compile-time approach may introduce more lock/unlock operations than strictly necessary, resulting in less concurrency.

Real-world programs make extensive use of libraries, hence being able to analyse them is important. However, libraries create a scalability challenge for static analysis [6] because they are large and have a high cyclomatic complexity.[1] Most problematic is that an analysis may not be able to complete if the memory requirements are too great. Furthermore, even simple programs can involve vast amounts of library code. Consider a "Hello World!" example 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. Analysing the library is a hard problem as is evident from the fact that existing work either ignores libraries [8–11], requires library implementers to annotate which method parameters should be locked prior to the call [12] or only considers accesses performed upto one level deep in library call chains [13]. All of these have the potential that some shared accesses performed within the library may go unprotected, leading to atomicity violations. Our previous approach [14] on this example was intractable.

Our main contribution is a lock inference approach for Java that analyses library methods in full. Specifically:

- We formulate lock inference as an Interprocedural Distributive Environment (IDE) dataflow problem.
- We adapt the pointwise graph representation of Sagiv et al [15] to reduce the number of edges in our summary graphs.
- We present *delta transformers* that dramatically reduce IDE analysis space-time requirements by only propagating new dataflow information.
- We identify and remove many locks for thread-local, internal, dominated and read-only objects.
- We implement our whole-program analyses in Soot.

We evaluate our approach as follows:

[1] Cyclomatic complexity [7] is a measure of the number of linearly independent paths. Library call chains can be long and consist of large strongly connected components.

- We demonstrate analysis scalability by analysing the entire GNU Classpath library (122KLOC) and the popular Java SQL database engine HSQLDB (150KLOC) on top of GNU Classpath.
- We evaluate the effects of a number of analysis optimisations: delta transformers, CFG summarisation [6], parallel processing of worklists and worklist ordering. We show that our delta transformers give the biggest speedup whilst also reducing memory usage.
- We evaluate the run-time performance of a range of benchmarks instrumented with our locks and compare results with the original synchronisation and Halpert et al [13].

2 General Approach

Our general approach is to use the Soot framework [16] to analyse Java classes annotated with atomic sections (we treat synchronized blocks and methods as atomic sections) and replace these annotations with suitable locks. Our analysis ensures weak atomicity.

```
1 class Scheduler {
2   Printer p1, p2;
3
4   atomic boolean schedule(Job j) {
5     // lockRead(this)
6     // lockWrite(this.p1);
7     // lockWrite(this.p2);
8     if (this.p1.job == null) {
9         this.p1.job = j;
10    } else if (this.p2.job == null) {
11        this.p2.job = j;
12    }
13    // unlockWrite(this.p2);
14    // unlockWrite(this.p1);
15    // unlockRead(this)
16  }
17 }
```

Fig. 1. An example atomic method and the locks we infer

First, we perform a dataflow analysis to infer what objects are accessed in each atomic section. Nested atomics are flattened and merged. We compute a summary for each method, which describes the accesses performed by it and all transitively called methods. The result is a graph describing all objects accessed in the atomic section, which we convert to locks.

We infer *instance locks* where possible, however, for those portions of the graph that describe a statically unbounded set of accesses, we infer locks on

the *types* of these objects. We use *multi-granularity locking* [17] to support both kinds of lock simultaneously: a type lock can be acquired if none of the locks on its instances are currently acquired and vice-versa.

We use simple analyses to identify objects that don't have to be locked such as thread-local or internal objects. We also detect when only a single thread is executing to avoid acquiring/releasing locks.

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 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.

Fig. 1 shows an 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` (which handles 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`.

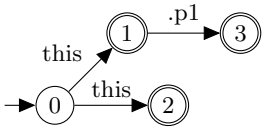
3 Inferring Accesses

In this section, we present our analysis for inferring which objects are accessed by each atomic section. We represent object accesses as syntactic expressions called *paths* [18, 14]. A path is an expression used to identify an object in code and consists of a variable followed by zero or more field and/or array lookups in any order. An example of a path expression is `x.f.g[i].h`. Our main contribution is that our analysis can scale to large programs that make use of large libraries.

We compute a summary function for each method m that describes the cumulative effects of m (including all methods transitively called by m). These functions are computed by composing the individual transfer functions for each of m's statements where the dataflow information are these transfer functions. For scalability, it is essential to have a compact representation for transfer functions with fast composition and meet operations.

For the general class of dataflow problems, called *Interprocedural Distributive Environment* (IDE), Sagiv et al [15] represent transfer functions as graphs, allowing composition to be computed by taking the transitive closure and meet by graph union or intersection. Rountev et al [6] have also shown that IDE analyses with this representation can scale well to programs using large Java libraries. We thus formulate our analysis as an IDE dataflow problem. In an IDE problem, dataflow values are mappings called *environments*. Transfer functions are called *environment transformers*.

In previous work [14], we represent sets of paths as *non-deterministic finite automata* (NFA). Fig. 2(a) shows an example NFA we might infer for the set {`this`, `this.pl`}. In our NFAs, numbers within states refer to the CFG node that generated the access. So, in this case, CFG nodes 1 and 2 generated an access of `this` and CFG node 3 generated an access of `this.p1` (whereby the `this` was generated at CFG node 1). We assume three-address code and thus each



(a)

this $\mapsto \{(0,1),(0,2)\}$
.p1 $\mapsto \{(1,3)\}$

(b)

Fig. 2. (a) Example NFA for the set of paths $\{this, this.p1\}$ and its environment representation (b)

CFG node dereferences at-most one object. Labelling NFA states with the CFG node that generated them allows us to efficiently detect looping accesses [14]. Transition labels correspond to variables, class names (for static lookups), field names and `[*]` to represent an array lookup.

IDE analyses require dataflow values to be maps, so in this paper, we represent NFAs as mappings of the form $\Sigma \to \mathcal{P}(Q \times Q)$, where Σ is the set of transition labels and Q is the set of NFA states. The states in each pair refer to the source and destination of the transition respectively.

3.1 IDE Transformers

We now define the environment transformers for our analysis. Transformers describe how dataflow values, i.e. environments, should be translated for a particular program statement. The challenge we face is 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, due to assignments that occur in-between. Our analysis is a backwards analysis because we push path expressions upwards. Our transformers translate these paths to preserve the set of objects that are accessed below, albeit potentially introducing new accesses due to the conservatism of our alias analysis (we use type information).

Fig. 3 contains our transformers, which we now describe in turn. We use Soot's three-address *Jimple representation*. We also assume a control flow graph (CFG) exists, whereby 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 was pointed-to by `y` before the assignment. To preserve object accesses performed lower down, paths beginning with `x` are rewritten to begin with `y`. 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: `y` $\mapsto e(y) \cup e(x)$, and deleting `x`'s transitions: `x` $\mapsto \emptyset$.

$[\texttt{x = new}]^n$ *and* $[\texttt{x = null}]^n$ 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: `x` $\mapsto \emptyset$.

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

Fig. 3. Environment transformers for path inference

$[x = y.f]^n$ 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. However, this assignment 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 we lock `y`. (ii) `x` and `z` are not aliases: the object `z` is not modified by the 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 the field f is the same as that being accessed 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 only adds the access of `x`.

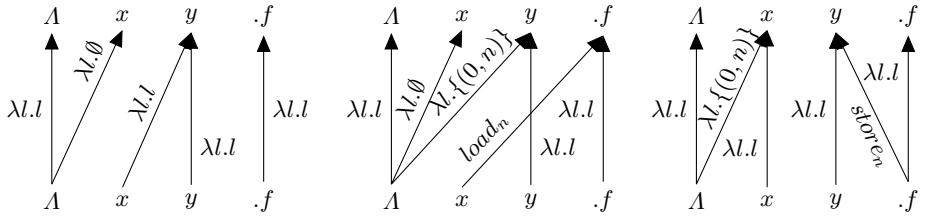
$[x = y[*]]^n$ The transformer for this statement is similar to that for `x = y.f`. We record the access of the array object `y` in the incoming environment e: $y \mapsto e(y) \cup \{(0, n)\}$. We do not distinguish between different array locations representing them all using [*], which can be read as "somewhere in the array." Our transformer preserves object accesses by translating all paths that begin with `x` to start with `y[*]`. 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, so this assignment could affect all paths that end in `[*]`. When translating, 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 $n'' \xrightarrow{[*]} n'$: $y \mapsto e(y) \cup \{(0, n') | (n'', n') \in e([*])\}$.

3.2 Graph Representation of Transformers

We now present the pointwise graph representations for our transformers. Informally, these graphs describe how the outgoing environment e' is derived from the incoming environment e when passing through a program statement. 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 $f : \mathcal{P}(\mathcal{Q} \times \mathcal{Q}) \to \mathcal{P}(\mathcal{Q} \times \mathcal{Q})$ 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 are introduced using the special symbol Λ.

Fig. 4 shows the pointwise representations for $t_{[x=y]^n}$, $t_{[x=y.f]^n}$ and $t_{[x.f=y]^n}$ from Fig. 3 (we assume that $\Sigma = \{x, y, .f\}$). Note: our analysis is backwards. Our analysis has five edge functions:



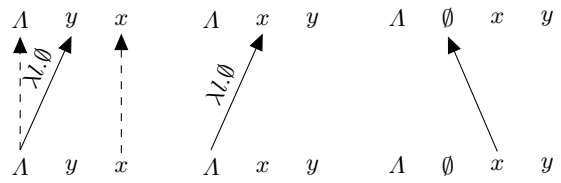
(a) $t_{[x=y]^n}$ (b) $t_{[x=y.f]^n}$ (c) $t_{[x.f=y]^n}$

Fig. 4. Pointwise representations for Fig. 3 key transformers

1. $\lambda l.\{(n', n'')\}$ for introducing a new automaton transition $n' \xrightarrow{d} n''$. For example, the statement $[x = y.f]^n$ of Fig. 4(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 Fig. 4(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 Fig. 4(a) collectively give that $e'(y) = e(y) \cup e(x)$ (as defined in Fig. 3).
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 $[x = y[*]]^n$. In Fig. 4(b), the edge $x \xrightarrow{load_n} .f$ rewrites all paths beginning with `x` to instead begin with `y.f` (the access of `y` is represented with the edge $\Lambda \xrightarrow{\lambda l.\{(0,n)\}} y$). The important thing to note is that n is common in both edges.
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 $[x[*] = y]^n$. In Fig. 4(c), $.f \xrightarrow{store_n} y$ adds an access of `y` for every path ending in `.f`. Existing paths ending in `.f` are preserved with the edge $.f \xrightarrow{\lambda l.l} .f$.

3.3 Sparsity

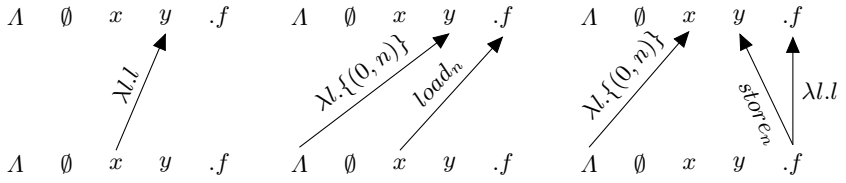
Sparsity is important to keep memory usage down. We keep graphs sparse by not explicitly representing trivial edges of the form $d \xrightarrow{\lambda l.l} d$. These implicit edges should not have to be made explicit, as that would be expensive. However, it turns out that determining whether an implicit edge exists is costly for our analysis. Fig. 5(a) shows an example transformer and Fig. 5(b) is its sparse equivalent. Dashed edges are used for trivial edges. Both x and y have no outgoing edges but while the implicit edge $y \xrightarrow{\lambda l.l} y$ exists, the same is not true for $x \xrightarrow{\lambda l.l} x$. This is because x is killed in the outgoing environment, as represented by the edge $\Lambda \xrightarrow{\lambda l.\emptyset} x$. To determine if an implicit 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.



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

Fig. 5. Determining whether an implicit edge exists is costly

To overcome this problem, we firstly introduce a new special symbol $\emptyset$. Killing the value for symbol d_i in the outgoing environment is then represented with the edge $d_i \to \emptyset$. Secondly, we observe that a large majority of our transformers perform kills, 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 $d_i \to d_i$ exists iff d_i has no outgoing edges. Fig. 5(c) shows the refined graph. Symbols Λ, $\emptyset$ and y have no outgoing edges and so each have implicit edges. x has an outgoing edge, therefore has no implicit edge. Fig. 6 shows the refined sparse pointwise representations of Fig. 4. In the case of Fig. 4(c), as we do not kill $.f$ in the outgoing 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 transformers contain significantly fewer edges.



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

Fig. 6. Refined pointwise representations for Fig. 4

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. Fig. 7(a) gives two example transformers whose meet is to be computed. The first transformer preserves all values from the incoming environment to the outgoing environment. The second transformer, however, copies x's value across to y before killing x's value. Hence, the combined transformer should both preserve x's value and also copy it to y. Fig. 7(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 an implicit edge explicit if at least one other transformer doesn't also have the implicit edge. If none of the transformers have the implicit edge, then it isn't generated in the merged result. The result for this example is shown in Fig. 7(c).

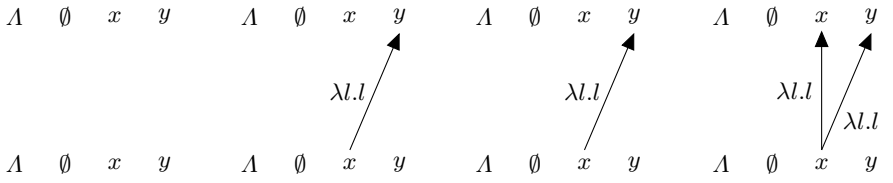


Fig. 7. Computing the meet when implicit edges are present

3.4 Native Code, Reflection and Dynamic Loading

All prior lock inference approaches, including our own [14], assume a closed world. Hence, they typically ignore native code, reflection and dynamic loading and assume that the only classes loaded are those that are analysed. We deal with native code/VM calls in this paper like Halpert et al by assuming all effects on the receiver and parameter objects. We do not handle reflection but could approximate the effects of reflective calls by using a tool such as TamiFlex [19], which executes a Java program and produces a reflection trace file. This trace can then be used by the analysis. It would not be possible to acquire locks late in dynamically loaded code, as it may lead to deadlock.

4 Inferring Locks

In this section, we describe how we convert NFAs computed by our analysis to locks. The important challenge here is to balance concurrency and locking overhead. We use instance locks where possible and revert to coarse type locks when we cannot statically determine the set of objects being accessed. We would like to acquire fine-grained locks when we can but still allow the possibility of acquiring type locks when necessary, so we use multi-granularity locking. We also describe how we identify objects that do not need to be locked.

From the summary function computed by our IDE analysis at the start of the atomic section, we extract the NFA describing all objects that may be accessed in the atomic section. We use our previous algorithm [14] to convert the NFA to instance and type locks with the modification that for this paper we use points-to information to determine the possible types of objects involved in cyclic accesses

(e.g. linked list traversal), rather than conservatively inferring all types in the class hierarchy rooted at the object's static type.

Our access inference analysis assumes that all object accesses need to be locked. However, there are some objects which do not need to be locked. We identify several classes of such objects: thread-local, instance-local, method-local, class-local, read-only and dominated objects. We also detect when there is only a single thread executing and avoid taking locks in this case. We now describe each of these.

4.1 Thread-Local Objects (TLA)

An object only needs to be locked if it may be accessed by multiple concurrent threads. We perform a simple analysis to identify objects that are thread-local and do not infer a lock for them. We use Lhotak [20]'s BDD-based thread local analysis implemented in Soot's Paddle framework. This analysis defines all objects reachable from static fields or fields in Runnable classes as being thread shared [34]. It uses Paddle's points-to graph to find these objects.

4.2 Internal Objects (ILA)

Another class of objects we avoid locking are internal objects that exist solely to implement the functionality of another object. An example is the underlying array object used in Java's `ArrayList` implementation. Such internal objects are dominated by their enclosing object `O`, meaning that all accesses to them are performed solely by `O`. This means that to protect accesses to them, when locking is performed outside `O`, it is sufficient to acquire a lock on `O`.

We use a simple and conservative flow-insensitive escape analysis to identify objects that are never accessed outside the instance they are created in. 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.
- The object was created in the application's `main()` method or a thread's `run()` method.

A field may become external if:

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

Initially, static fields are marked *External*, instance fields are marked *Internal*, non-static method parameters are *Internal* and static method parameters are *External*. We model the return value as assignment to a special return variable `r`,

which is initially *Internal* for instance methods and *External* for static methods. For all methods, `this` is always *Internal*. We model array lookups as fields.

Our whole-program analysis finds all reachable methods in the program (including all reachable library methods) and processes them sequentially until a fixed-point is computed. We do not process the call graph in any particular order. 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 the return value. Our analysis can also handle inner classes (as used by iterators) and object handover, such as `A a = new A(new B());` (here the new instance of `B` is being handed-over to the new instanceof `A`).

We use the results of our escape analysis when converting the access NFA to locks by locking the outermost object to protect accesses of internal objects. We can handle multiple levels of internal objects within a single outermost object.

4.3 Single-threaded Execution

We have found that during the initialisation of an application, many objects are accessed but there is typically only a single-thread executing. Lock acquisitions and releases of our locks can impose significant overheads in this scenario (we do not use thin locks). Thus, we optimise our lock implementation so that locks are treated as no-ops when there is only one thread executing. These object accesses are not thread-local but just that they are only being accessed by a single thread at present. We already remove thread-local locks, as described above.

We detect whether only a single thread is executing or not by incrementing and decrementing a counter when `Thread.start()` and `Thread.join()` are called respectively. If this counter is 0 then we elide the locks otherwise we acquire them as normal. This works because we assume that threads are not spawned by atomic sections.

4.4 Multi-granularity Locks

We use the multi-granularity locking protocol of Gray et al [17] to simultaneously support both type and instance locks. Usually, both coarse- and fine-grained locks cannot protect the same data simultaneously so only one of them would be used. However, multi-granularity locking allows both to be used at the same time for the same data. The multi-granularity locking protocol allows an instance lock to be taken if a coarse-grained type lock protecting the same object hasn't already been acquired and vice-versa. When locking a large number of objects, such as all instances of a type, one can reduce locking overhead by locking the type, whereas in other cases one can lock individual instances to get more concurrency. This orchestration is done at run-time. We implement [21] these locks using Doug Lea's Synchronizer framework [22, 23] for performance.

4.5 Other Optimisations

In addition to removing thread-local and instance-local locks, we perform a number of optimisations to further reduce the locks inferred. This includes analyses for finding: locks that are dominated by other locks (DOM), locks for method-local objects (MLA), locks for objects referred to by static fields that never escape the enclosing class and can therefore be protected by locking the corresponding `Class` object (CLA), objects that are only ever locked in read-mode and are thus read-only (RO) and finally, types that do not need to be locked in intention mode [17] when their instances are locked (IMP).

Many of these analyses require looking at locks across all atomic sections (e.g. to find out which objects are read-only), therefore we did a final optimisation to ignore atomic sections that are not-reachable from the program's `main` method and will thus never be executed. This greatly improved the results of the previous mentioned analyses.

4.6 Deadlock

Atomic sections must not deadlock as a result of the locks we insert. We acquire locks at the start of the atomic section, allowing us to prevent deadlock at run-time and thus keep per-instance locks, rather than coarsen the locking granularity or impose a static ordering at compile-time and thus potentially hinder concurrency [13, 12]. Given that deadlock rarely occurs, such compile-time approaches to deadlock-avoidance are undesirable.

We avoid deadlock at run-time as follows: when a thread is about to block on a lock l, it first releases all already acquired locks. It then blocks waiting for l to become available after which it starts from scratch to try to acquire all locks (starting from the first lock). This guarantees freedom from deadlock, as it means that locks are not held while waiting, thereby breaking one of the four necessary conditions for deadlock [24].[2] Path expressions must be re-evaluated when re-acquiring locks after waiting because there may have been concurrent updates to the heap. This approach to avoiding deadlock is essentially the same as *retry* used in STM [25]. To improve performance, we use an adaptive locking scheme whereby we first poll l N times before releasing all already acquired locks. Thereafter, we don't block waiting on l but poll until it becomes available before starting the locking stage from scratch as mentioned above.

5 Implementation

We implemented our lock inference approach as a whole-program transformation in Soot (SVN r3588). Here, we give details including optimisations to reduce the memory consumption and running time of our analysis.

[2] We reduce the likelihood of livelock by using random backoffs.

5.1 Summary Computation

In this section we describe how we compute per-method summaries. Each method m has a unique entry node N_m and exit node X_m. We store for each CFG node n in m, its local transformer t_n that describes how n transforms environments (see Fig. 3) and an aggregate transformer t_{n,X_m} that summarises the transformation on environments along all execution paths between n and X_m inclusive. The local transformer for a method invocation statement $[x = y.foo(a_1, ..., a_k)]^n$ encapsulates three steps: (i) parameter passing, (ii) invocation of the callee method *foo* and (iii) storing the return value to result variable x. Thus, t_n can be expressed as $t_n = t_{n_{params}} \circ t_{n_{invoke}} \circ t_{n_{result}}$. The transformer $t_{n_{invoke}}$ is the summary of the callee *foo*, i.e. T_{foo}. However, due to polymorphism, there may be several possible callees. We therefore take the meet of all such callee summaries.

The summary T_m for a method m is 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 the current strongly connected component. Either list is processed exclusively until it becomes empty because interprocedural propagation is expensive and hence it is more efficient to do as much intraprocedural propagation as possible before propagating across method boundaries.

5.2 Reducing Space and Time Requirements

There can be many CFG nodes and transformers can get very large, leading to vast memory usage and slow analysis times. We employ the following techniques to reduce both memory and running time.

Delta Transformers 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 leads to redundant work because (i) transformer composition is distributive, hence if two edges (one from each transformer) have already been composed before, composing them again will give the same result and (ii) taking the meet is union and unioning old edges gives nothing new. The distributive nature of the analysis thus allows us to process only new edges and then union the results with what has already been computed. We differentiate new edges using a different type of transformer, which we call *delta transformers*. This approach of only propagating additions gives us the biggest speed up and the second best reduction in memory usage.

Summarising CFGs We implement the technique of Rountev et al [6] that summarises the effects of all execution paths between a pair of CFG nodes n_1 and n_2 in a method m, by combining transformers for statements along these paths. This summary $t_{n_1 \to n_2}$ allows dataflow information to be propagated from n_2 to n_1 (backwards analysis) in one step by composing with it thus reducing propagation and storage. The result of this optimisation is a reduced CFG for m containing three types of nodes: N_m, X_m and recursive calls rc_i together with a set of summary transformers describing effects along execution paths between them.

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 successor has changed. Although an ordering exists between CFG nodes in the same method, we can exploit the independence between different methods to construct a set of per-method worklists and process the lists in parallel. Our inter worklist contains call nodes that need to be updated when 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 call nodes in the list so we process them all in parallel.

Efficient Data Structures Efficient implementations typically use primitives to represent state [22] 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), which again is not ideal. We use the Trove library[3], which provides primitive implementations of many data structures such as `HashSet`s and `HashMap`s. We implement transformers as maps, using integers to represent symbols (and sets of longs for their edges).

Worklist Ordering Ordering the worklist so that successor CFG nodes are given preference over predecessor nodes is an important and well-known optimisation. This makes intuitive sense because dataflow information propagates up the CFG, so if a successor is to be processed again (i.e. it is in the worklist), it may as well be processed before its predecessors in case its value changes once more, thus avoiding unnecessary propagation. We were surprised that this optimisation gave a bigger speed up than CFG summarisation.

6 Evaluation

We now present experimental results for our lock inference approach. We used two experimental machines: (1) *liatris*: a commodity machine consisting of an 8-

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

core 3.4GHz Intel Core i7-2600 CPU, 8GB RAM and running Ubuntu 11.04; and (2) *ax3*: a much larger machine containing 32 8-core 2.67GHz Intel Xeon E7-8837 CPUs totalling 256 cores, 3TB RAM and running SUSE Linux Enterprise Server 11. We use ax3 for analysing `hsqldb` and liatris for analysing all other code and for executing all programs. For running our analysis, we used Oracle's 64-bit JVM and for instrumented runs, we used a modified version of the Jikes RVM. On liatris, we used Oracle JVM version 1.6.0_26-b03 with a minimum/maximum heap size of `4GB` and version 1.7.0_03-b04 with a minimum/maximum heap size of `70GB` on ax3. We did not specify a stack size in either case. For running programs, we used a modified version of the production build of Jikes RVM version 3.1.1+svn (r15989M). Details of the modifications we made are given below.

We begin by giving results for Hello World and use it as a basis for comparing the effect of our different analysis optimisations, as described in Section 5. We demonstrate in Sections 6.4–6.5 that our analysis techniques can scale to large programs by analysing the GNU Classpath library (122KLOC) and the Java database engine hsqldb (150KLOC). Note, for Hello World and GNU Classpath, we used a minimum/maximum heap size of `10GB`.

We evaluate the run-time performance of the benchmarks *sync*, *pcmab*, *bank*, *traffic*, *mtrt* and *hsqldb* instrumented with our locks and compare results with Halpert et al [13], the benchmark's original synchronisation as well as using a single global lock in Section 6.6. Our running times are for all lock optimisations enabled (thread-local, internal object, dominators, etc.). Furthermore, for a fairer comparison, we replace the original `synchronized` blocks and methods with our locks (albeit still maintaining the same locking policy and behaviour as the original `synchronized` blocks).

For fast instance lock retrieval, we modify Jikes by adding an `ilock` field to every object. For fast lookup of type locks, we extend `java.lang.Class` with a `tlock` field. We minimise the additional overhead to object creation by lazily instantiating the lock field. While using a lock table would avoid this, it introduces a lookup overhead which is encountered every time the lock is required. We also extend the `Thread` class for quick access to thread local data (as opposed to using `java.lang.ThreadLocal` that incurs high overhead).

6.1 Soundness of Halpert et al

Halpert et al [13] analyse library call chains upto one level deep and rely on original library synchronisation beyond that. There are many programs where this is sufficient, but it is not sufficient for all problems. Code which has deep library calls fails. Furthermore, if there is no manual synchronisation present then their approach does not guarantee safety of library accesses. For instance, we ran their tool (r3043) on the Hello World program, having removed the existing synchronisation in the library[4] and observed that because they only

[4] See http://www.doc.ic.ac.uk/~khilan/code/ConcurrentPrintln.java.txt for the program code

(a) **Analysis (secs)**			(b) **Locks**			
			Instance		*Type*	
Paths	Locks	Total	Read	Write	Read	Write
33	0.6	47	215	54	148	34

Fig. 8. Analysis results for Hello World

analyse one level deep they inferred empty read and write sets and when the program executed, print buffers were corrupted causing strings to be printed out multiple times or not at all.[5] Any comparison with their work is a loose one but we do so because it is the closest work to ours.

6.2 Hello World

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 [14], we found it intractable.

The running times (in seconds) for the path and lock inference analyses are given in Fig. 8(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 times reported are with all analysis optimisations enabled and 8 worker threads. The number of instance read, instance write, type read and type write locks inferred are given in Fig. 8(b). We do not remove any locks. Memory usage peaks at 3.1GB and averages 1.6GB.

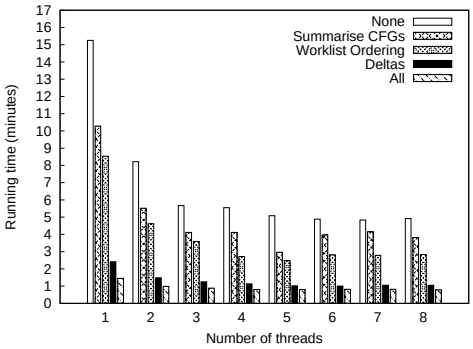
6.3 Analysis Optimisations

In this section we evaluate the impact of the analysis optimisations from Section 5.2. We use the Hello World program and compare the effects of delta transformers, CFG summarisation, worklist ordering and parallel propagation on memory usage and running time. All configurations uses the efficient data structures detailed in Section 5.2. Fig. 9 shows our comparison.

The comparison gives 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 even if only one thread is used. 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 lesser work over time. Memory usage is also lower because temporary objects are reduced.

Also, 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 in sequence

[5] The output of running their tool on Hello World can be found on http://www.doc.ic.ac.uk/~khilan/code/ConcurrentPrintlnHalpertOutput.txt



(a)

Optimisation	Average MB	Peak MB
None	4923.92	8183.18
Summarise CFGs	2094.68	3470.65
Worklist Ordering	4804.73	8037.14
Deltas	3848.98	6538.27
All	1741.39	3122.84

(b)

Fig. 9. Effect of each optimisation on analysis time (a) and memory usage (b) for Hello World

(a) **Library Info**		(b) **Analysis (secs)**		(c) **Locks**			
				Instance		*Type*	
Package	Methods	Paths	Locks	R	W	R	W
gnu	16882	64.43	9.40	16536	6235	7510	1310
java	13815	46.11	13.67	30065	9940	30007	5354
javax	14088	11.06	6.03	7640	3307	0	0
org	2794	1.31	1.09	1275	401	0	0
sun	28	0.01	0.03	11	4	0	0
Total	47607	127.79	30.22	55527	19887	37517	6655

Fig. 10. Analysis results for GNU Classpath 0.97.2

(we do not start processing inter until there is nothing left to do in intra and vice-versa). Consequently, threads that have become free cannot proceed with the other list until all threads have completed. Some methods may require more propagation than others and so this creates a bottleneck.

We were surprised that ordering the worklist so that successors are given preference over their predecessors outperformed the running time when summarising CFGs. This might indicate that unnecessary propagation occurs quite often if worklists are not ordered appropriately.

6.4 GNU Classpath

To evaluate the scalability of our path analysis, we analyse the entire GNU Classpath 0.97.2 library. It consists of 47607 non-private methods and totals about 122KLOC. We analyse each of these non-private methods in turn[6], treating it as an atomic method. We re-use summaries if they have been computed already (during the current analysis run).

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

We ran our analysis with all analysis optimisations turned on and with 8 worker threads. It took 5 minutes and produced a summary file of size 381MB. Memory usage peaks at 5.1GB and averages 3GB. Fig. 10 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. Again, we do not remove any locks.

The method which took the longest to analyse was `Logger.getLogger(String)` (30 seconds). Upon inspection, we found that this pulled in the same part of the library as Hello World. Once this set of methods had been analysed, the summaries for methods called by most other methods had already been computed and so did not have to be recomputed. The remaining methods were analysed in a fraction of the time (average of 2ms).

From the locks inferred (Fig. 10(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).

6.5 HSQLDB

Large real-world programs make extensive use of libraries. We evaluate how well our approach can handle one such program: hsqldb.

This is an SQL relational database engine providing both in-memory and disk-based tables. It is widely used in many open-source as well as commercial products. We use the benchmark version (1.8.0_4) packaged in the Dacapo benchmark suite [26], consisting of an in-memory banking database against which a number of transactions are executed. It comprises a total of 150KLOC and 240 atomic sections (we treat synchronized blocks and methods as atomic sections), as well as making extensive use of GNU Classpath. Fig. 11(a)(i) gives a breakdown of the total number of client and library methods called by atomic sections. Of the 5062 methods called, 58% are in the library.

Our path 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 GC. 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. Our lock-removing analyses were able to identify many locks that could be removed, as shown in Fig. 12(a).

Program	Threads	Atomics		(i) Methods		(ii) Analysis (secs)		(iii) Run (secs)			
		Total	Reachable	Client	Library	Halpert	Ours	Manual	Global	Halpert	Ours
sync	8	2	2	0	0	22	127	69.14	71.22	72.69	56.61
pcmab	50	2	2	2	15	22	127	2.28	3.15	2.28	2.47
bank	8	8	6	6	7	22	127	20.89	19.50	35.69	3.88
traffic	2	24	19	4	63	24	130	2.56	4.22	2.65	4.42
mtrt	2	6	4	67	1324	29	169	0.80	0.82	0.78	0.85
hsqldb	20	240	158	2107	2955	48104	23886	3.25	3.12	3.25	11.39

(a)

Program	Paths (secs)	Locks (secs)	Lock optimisations (secs)						
			TLA	ILA	DOM	CLA	RO	IMP	MLA
sync	0.053	0.0090	0.598	8.441	1.42	3.979	0.0010	0.0	0.0010
pcmab	0.194	0.018	0.603	8.309	1.444	3.855	0.0010	0.0	0.0020
bank	0.151	0.019	0.408	8.177	1.376	3.802	0.0020	0.0010	0.0020
traffic	0.433	0.059	0.569	9.267	1.625	3.861	0.0060	0.0020	0.465
mtrt	33.901	1.902	0.623	9.063	1.741	4.259	0.079	0.03	0.0050
hsqldb	21936.024	1345.859	1.667	28.589	9.597	53.125	1.84	2.724	0.079

(b)

Fig. 11. Analysis and run-time results comparison for a selection of benchmarks from Halpert et al [13, 27]. (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 (before optimisations).

6.6 Comparison with Halpert et al

We compare the running times of a selection of benchmark programs transformed using our approach with the closest known existing work of Halpert et al [13] in Fig. 11(a).[7] 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).

An important difference between our approaches is that we analyse library methods in full whereas they 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 6.1). In Fig. 11(a)(i), we list the number of client and library methods called by atomic sections. Fig. 11(a)(ii) compares analysis times (both columns include Soot-related costs). We give a breakdown for the running time of each component in our analysis in Fig. 11(b).

Fig. 12(a) gives a comparison of locks inferred. Fig. 12(a)(i) are the locks inferred by Halpert et al, Fig. 12(a)(ii) the locks we infer and Fig. 12(a)(iii) again shows the locks we infer but this time after applying all our lock optimisations. We give a breakdown of how many locks are removed by each respective lock

[7] We do not use their published work [13] but their later improved version [27] 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.

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

(b)

Fig. 12. Locks inferred for benchmarks in Fig. 11 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 of our lock optimisations.

optimisation in Fig. 12(b). The number for IMP indicates how many instances do not need to intentionally lock their type.

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.

Fig. 11(c) gives execution times. We are noticeably slower for *hsqldb* due to the larger number of locks being acquired. Note, *hsqldb* involves a large number of library methods, which are not analysed by Halpert et al so a direct comparison is not appropriate. At run-time, only 2745 of the 5062 methods (54%) analysed for hsqldb are called. We are looking into using run-time coverage information to reduce the number of locks taken for code paths that are infrequently executed.

7 Related Work

Lock Inference While software transactional memory remains the popular approach for implementing atomic sections, recent work has also looked at statically inferring locks sufficient for atomic and deadlock-free execution.

In McCloskey et al's Autolocker tool [12], the programmer annotates which locks protect each path expression. Locks are acquired before object accesses and released at the end of the atomic section. Deadlock is prevented statically

by ordering path expressions for locks, with the program being rejected if an ordering is not possible. This approach is shown to scale to a 50KLOC web server. Autolocker allows internal objects to be protected by the same lock with a suitable annotation, however our approach differs because we automatically infer these objects. Emmi et al [10] extend upon Autolocker by removing the need for annotations. They have two types of lock: per-instance and per-path expression whereby the latter protects all instances of a path expression and is used when two path expressions p_1 and p_2 may alias each other along some execution path. Lock inference is formulated as an 0-1 ILP problem that aims to minimise the number of locks as well as the number of conflicts between atomic sections. Their approach is shown to scale to 15KLOC. These two approaches do not translate path expressions past assignments and subsequently lock operations cannot be pushed further up without coarsening the locking granularity.

Hicks et al [8] infer *abstract objects* that are each protected by their own lock. This has the advantage that deadlock can be prevented statically, as the number of locks is known at compile-time. However, less concurrency may occur because per-instance locks are not supported. Locks are acquired at the start and released at the end of the atomic section. We base our dominators analysis on theirs but ours differs because we do it for path expressions. Furthermore, we have a working implementation.

Cherem et al [9] also infer path expressions and translate them when pushing through assignments. They also acquire locks at the start of the atomic and release them at the end. However, there is a major difference: we represent paths as non-deterministic finite state automata, allowing unbounded accesses to be represented precisely, whereas they immediately collapse these accesses to some lock R. The focus of their work is a general theoretical framework for lock inference analyses.

Finally, Halpert et al [13] and Zhang et al [11] take a top-down approach (whereas those previously mentioned and this paper are bottom-up approaches) by instead determining which atomic sections may conflict with each other and then preventing them from proceeding in parallel by allocating to each an appropriate set of locks. These locks may or may not have any relation to the objects being accessed but their purpose is just to prevent conflicting atomics from running concurrently. Bottom-up approaches, like ours, map accesses to locks, which implicitly prevent conflicting atomics from running in parallel. Halpert et al use a May Happen In Parallel [28] analysis to improve the precision of conflict detection and a thread-local/thread-shared analysis to reduce the size of the read/write set of each atomic section. They do not require two-phased locking. However, deadlock is prevented by assigning the same static lock to each atomic section involved in the wait-cycle, thus preventing them from executing in parallel, even if on some/most runs they could. Halpert et al analyse Java programs but only analyse library call chains upto one level deep.

Interprocedural Analysis of Large Programs The original *callstrings* approach for interprocedural analysis [29] is known to not scale well [30]. Khedker et al [30] propose grouping callstrings into equivalence classes based upon dataflow

values and subsequently performing propagation through a method only once per value. We implemented this but were not successful for Hello World. However, this may be due to our implementation of the technique. Regardless of this, the callstrings approach also has the limitation of not allowing pre-computed results for a method to be stored for re-use later, as it does not encode how methods translate dataflow information. Consequently, library methods would have to be re-analysed at each call site.

Recent work [15, 31, 32, 6, 33, 34] has looked at scalable interprocedural analyses using procedure summaries. [15] present an efficient graphical representation for transfer functions and [6] apply this to Java programs that make use of the library. Our work differs from [6] as they present a general framework for whole-program IDE analyses but do not apply it specifically to lock inference . We assume a closed world whereas they consider the possibility of call backs from the library to client code. Our work could be extended to cater for this.

While propagation of delta information is not a new idea, we believe that this is the first time that they have been presented for the IDE framework.

8 Conclusion & Future Work

We believe that this is the first lock inference approach that can analyse precisely programs built with large libraries. Previous lock inference work [14, 11, 13, 12, 8–10] either ignores libraries, requires library implementors to annotate which locks to take or only consider accesses performed upto one level deep in library call chains [13]. We are able to handle large programs by formulating our previous path inference analysis [14] as an IDE dataflow problem. We have shown that our analysis can scale to 122KLOC when using the pointwise representation of [15, 6] together with a number of optimisations, which in turn we have evaluated. We also analysed the large Java database engine HSQLDB comprising 150KLOC (plus 3000 methods from GNU Classpath).

We have also implemented several analyses to reduce locks inferred, such as for thread-local and internal objects. Our lock inference approach is the first to automatically identify internal objects and elide locks for them. Furthermore, these analyses are conservative but scale to library code and are still able to identify many such objects, which we have shown through the hsqldb benchmark. We detect when only a single thread is executing and elide locks in this case too. This is orthogonal to thread locality because these are locks for shared objects. This reduced our run-times tremendously, as the locking overheads incurred during single-threaded execution were mitigated.

We evaluate the run-time performance of our instrumented programs for a range of benchmarks and compare results with Halpert et al [13]. Halpert et al 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 6.1).

In this kind of work there are always ways to improve it. We believe that major areas of a program may rarely be executed and are looking to take advantage of this to reduce the number of locks taken at run-time by delaying the acquisition of locks protecting such cold code regions.

We don't expect our run-times to match those of optimal hand-crafted locks, however for most code they are probably acceptable. More importantly it should be a far simpler task for programmers to annotate blocks of code as atomic than to get them to place locks correctly, and a correctly annotated program will be a deadlock free, race free program.

Acknowledgements We are grateful to Microsoft for funding this work. We would like to thank Dave Cunningham for the original idea [14] and the belief that reasonable results could be obtained. We are also very appreciative of the detailed discussions we had with Tristan O. R. Allwood and Sophia Drossopoulou and all their helpful advice. We thank Richard Halpert for providing his benchmark programs and scripts. We also thank the entire SLURP research group at Imperial College for interesting discussions about earlier versions of this work. The work would not have been possible without the advice of members of the Soot, Jikes RVM and concurrency-interest mailing lists.

References

1. Lomet, D.B.: Process structuring, synchronization, and recovery using atomic actions. SIGPLAN Not. (1977)
2. Grossman, D.: The transactional memory/garbage collection analogy. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. (2007)
3. Cantrill, B., Bonwick, J.: Real-World Concurrency. ACM Queue (2008)
4. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal (2005)
5. Larus, J., Rajwar, R.: Transactional Memory (Synthesis Lectures on Computer Architecture). Morgan {& Claypool Publishers (2007)
6. Rountev, A., Sharp, M., Xu, G.: IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In: CC. (2008)
7. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. (1976)
8. 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). (2006)
9. Cherem, S., Chilimbi, T.M., Gulwani, S.: Inferring locks for atomic sections. In: PLDI. (2008)
10. Emmi, M., Fischer, J.S., Jhala, R., Majumdar, R.: Lock allocation. In: POPL. (2007)
11. Zhang, Y., Sreedhar, V.C., Zhu, W., Sarkar, V., Gao, G.R.: Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections. In: LCPC. (2008)
12. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. ACM SIGPLAN Notices (2006)

13. Halpert, R.L., Pickett, C.J.F., Verbrugge, C.: Component-Based Lock Allocation. In: PACT. (2007)
14. Cunningham, D., Gudka, K., Eisenbach, S.: Keep Off the Grass: Locking the Right Path for Atomicity. In: CC. (2008)
15. Sagiv, Reps, Horwitz: Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. TCS: Theoretical Computer Science (1996)
16. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON. (1999)
17. Gray, J.N., Lorie, R.A., Putzolu, G.R.: Granularity of locks in a shared data base. In: VLDB '75: Proceedings of the 1st International Conference on Very Large Data Bases. (1975)
18. Chan, B., Abdelrahman, T.S.: Run-Time Support for the Automatic Parallelization of Java Programs. J. Supercomput. (2004)
19. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In: ICSE '11: International Conference on Software Engineering. (2011)
20. Lhotak, O.: Program Analysis Using Binary Decision Diagrams. PhD thesis
21. Gudka, K., Eisenbach, S.: Fast Multi-Level Locks for Java: A Preliminary Performance Evaluation. In: EC$\hat{2}$ 2010: Workshop on Exploiting Concurrency Efficiently and Correctly. (2010)
22. Lea, D.: The java.util.concurrent Synchronizer Framework. Sci. Comput. Program. (2005)
23. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley (2006)
24. Magee, J., Kramer, J.: Concurrency: state models {& Java programs. Wiley New York (2006)
25. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (2005)
26. Blackburn et al, S.M.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. (2006)
27. Halpert, R.L.: Static Lock Allocation. Master's thesis, McGill University (2008)
28. Naumovich, G., Avrunin, G.S.: A Conservative Data Flow Algorithm for Detecting All Pairs of Statement That May Happen in Parallel. In: SIGSOFT FSE. (1998)
29. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. (1981)
30. Khedker, U.P., Karkare, B.: Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In: CC. (2008)
31. Gulwani, S., Tiwari, A.: Computing Procedure Summaries for Interprocedural Analysis. In: European Symp. on Programming, ESOP 2007. (2007)
32. Rountev, A.: Component-Level Dataflow Analysis. In: International SIGSOFT Symposium on Component-Based Software Engineering. (2005)
33. Whaley, J., Lam, M.: An efficient inclusion-based points-to analysis for strictly-typed languages. Proceedings of the 9th International Static Analysis Symposium (2002)
34. Choi, J., Gupta, M., Serrano, M., Sreedhar, V., Midkiff, S.: Escape analysis for Java. Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (1999)