Lock Allocation *

Michael Emmi Jeffrey S. Fischer
UC Los Angeles
mje,fischer@cs.ucla.edu

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Rupak Majumdar
UC Los Angeles
rupak@cs.ucla.edu

Abstract

We introduce *lock allocation*, an automatic technique that takes a multi-threaded program annotated with *atomic* sections (that must be executed atomically), and infers a lock assignment from global variables to locks and a lock instrumentation that determines where each lock should be acquired and released such that the resulting instrumented program is guaranteed to preserve atomicity and deadlock freedom (provided all shared state is accessed only within atomic sections). Our algorithm works in the presence of pointers and procedures, and sets up the lock allocation problem as a 0-1 ILP which minimizes the conflict cost between atomic sections while simultaneously minimizing the number of locks. We have implemented our algorithm for both C with pthreads and Java, and have applied it to infer locks in 15K lines of AOLserver code. Our automatic allocation produces the same results as hand annotations for most of this code, while solving the optimization instances within a second for most programs.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features.
General Terms: Languages, Algorithms.
Keywords: atomicity, lock inference, ILP.

1. Introduction

Shared memory concurrency, where multiple threads access shared data structures, is a pervasive programming model for multiple interacting computational tasks. Accessing shared data concurrently introduces the possibility of synchronization errors, which occur when the interleaved execution of multiple threads causes the assumptions by a thread on the shared state to be modified unpredictably. Correct programs must therefore ensure non-interference at program points that access shared state. The non-interference assumptions are formalized through the notion of *atomicity* [8, 7]. If a piece of code is atomic, then any interaction between that code and the steps of all other threads is guaranteed not to change the observable behavior of the program: for every possibly interleaved execution, there is an equivalent execution where the atomic code executes sequentially.

Atomicity is an application-level specification. In practice, the programmer intent of atomicity is often ensured through the mechanism of mutual exclusion monitors or *locks*. The programmer explicitly acquires and releases locks to synchronize access to shared structures while trying to maintain maximum parallelism between tasks. When correctly employed, locks ensure shared data accesses occur atomically. However, locking is notoriously difficult to get right—mistakes can lead to data races and non-atomic executions, or result in over conservative lock placement, which may significantly diminish the benefits of concurrency or even cause deadlock.

We present *lock allocation*, an automatic algorithm that takes a program annotated with atomicity specifications, and *infers* a suitable set of locks together with their placement in the code such that with this lock placement, the resulting code is guaranteed to preserve atomicity and prevent deadlock (under the assumption that all global variables accessed within atomic sections are only accessed within atomic sections). The lock allocation algorithm must balance the tension between ensuring atomicity (for example by maintaining one global lock that is acquired before each atomic section and released at the end) and maximal concurrency (for example by maintaining a different lock for each global variable and acquiring locks for all variables accessed in an atomic section) and low cost (each lock variable, acquire, and release operation assumes a certain cost). We set up the problem as a constraint optimization problem. In particular, we designate a boolean variable a_{ij} to be true if the ith global variable is assigned the jth lock, and set up constraints to ensure each global variable is assigned some lock. We then attempt to minimize the conflict cost (a measure of the loss of concurrency incurred if an atomic section is waiting on a lock currently held by another atomic section) between atomic sections, and the number of locks used, simultaneously. The resulting optimization problem is a 0-1 ILP, for which extremely efficient solvers (through reduction to boolean SAT) are available [4]. The solution is a *lock assignment* that assigns a lock to each global variable. As we demonstrate, the lock allocation decision problem is NP-complete, hence the reduction to 0-1 ILP is asymptotically optimal, and as our experiments demonstrate, practically efficient. Given a lock assignment, our placement algorithm is similar to Autolocker [14], a system that takes both atomic section annotations and lock assignments, and places locks to ensure atomicity and deadlock freedom. Our optimization scheme can be seen as *inferring* the Autolocker lock assignment annotations.

We extend the basic lock allocation algorithm to deal with pointers and functions. In the presence of pointers, we use a statically computed points-to relation, and add additional constraints to ensure that whenever two program lvalues are aliased at run time, their assigned locks must also alias at that point in the execution. Further, we add constraints that ensure that for every lvalue, its assigned lock is in scope when the lvalue is accessed. Additionally, we use a statically computed *dependency relation* that places an edge between two abstract locations l_1 and l_2 if there is an atomic section that accesses l_1 before l_2 along some program execution.

* This research was sponsored in part by the research grants NSF-CCF-0427202, NSF-CNS-0541606, and NSF-CCF-0546170.

POPL'07 January 17–19, 2007, Nice, France.

The dependency relation is then used to put in additional constraints that remove cyclic dependencies between locks (that can lead to deadlocks).

We have implemented the lock allocation scheme both for C programs using the pthreads thread library and for Java programs. For our experiments, we have used about 16K lines of code from the AOLserver web server, that was annotated with atomic sections by the developers of Autolocker, and inferred lock assignments for the variables. Our inferred lock assignments are (almost always) identical to the manual locking inserted by Autolocker. In some of these examples, one can obtain better locking manually by realizing certain lvalues cannot be aliased at run time. Since we use a static flow insensitive points-to and dependency relation, our algorithm is conservative. Increasing the precision of the analysis is an interesting orthogonal problem. However, as our experiments demonstrate, for many useful programs, programmers use simple locking disciplines that can be automatically inferred.

Related Work. In the programming languages context, [7] introduced type and effect systems to check atomicity in Java programs. While we use *pessimistic* or synchronization based schemes to ensure atomicity, there is a lot of work on ensuring atomicity through optimistic concurrency control that uses a combination of logging and rollback [8, 9, 12, 18]. The problem of race and deadlock detection has received a lot of attention from static/dynamic analysis [5, 19, 20] and model checking [11] communities.

Recent work has focused on either inferring atomic sections given lock assignments [6], or given atomic sections and lock assignments, instrumenting the program with locks to ensure atomic and deadlock free execution [14]. Our work is dual to [6]: given atomicity specifications, we infer locking mechanisms to ensure atomicity. Independently, [10] also considers the lock allocation problem. Unlike our optimization-based algorithm, their algorithm considers the points-to graph, assigning locks to abstract locations and unifying lock names based on a dependency relation on abstract locations. It is difficult to incorporate *quantitative* cost measures (measuring the cost to acquire/release a lock as well as conflict costs) in their scheme. Further, their scheme only produces *global* locks, whereas (as we illustrate later), our optimization framework can actually infer locks local to structures (that are instantiated with a different lock per instance of the structure) in some cases. Thus, our algorithm can be more precise.

2. Lock Optimization

We introduce the lock allocation scheme on multi-threaded programs with global integer variables. In the next section, we additionally handle references and aliasing.

Multi-threaded Programs and Atomicity. For the remainder of this section, we fix a set $X = \{x_1, \ldots, x_k\}$ of global integer variables in a multi-threaded program P. We assume that a set of code blocks $\mathcal{A} = \{A_1, \ldots, A_n\}$ of P are annotated as *atomic sections* by the programmer. For each $i \in \{1, \ldots, n\}$, let $\mathsf{access}(A_i) \subseteq X$ be the set of global variables accessed in block A_i.

The intent is that each atomic section *executes atomically* in P. Intuitively, when a block b executes atomically in P its interaction with other threads does not change the overall program behavior: for every interleaved execution of P in which b is executed, there is an equivalent execution where b executes uninterrupted by the other threads [7, 8].

Lock Allocation. Atomicity is typically ensured by protecting accesses to shared variables with locks. Given a set of locks L, a *lock assignment* is a mapping, $\lambda : X \to L$, assigning a lock to each shared variable. For an atomic section $A_i \in \mathcal{A}$, the set $\mathsf{locks}_\lambda(A_i)$ of locks required by A_i is the set of locks assigned to all variables accessed in A_i:

$$\mathsf{locks}_\lambda(A_i) = \bigcup_{x \in \mathsf{access}(A_i)} \{\lambda(x)\}. \quad (1)$$

To ensure atomicity of A_i, P acquires each lock $\ell \in \mathsf{locks}_\lambda(A_i)$ before the first variable protected by ℓ is accessed, and only releases ℓ upon exiting A_i.

The use of locks to protect shared variables may induce conflicts between atomic sections acquiring the same lock: the execution of an atomic section will be delayed until the locks it requires are released by the atomic sections of other threads. For each pair A_i, A_j of atomic sections, let $\mathsf{cost}(i, j)$ be a penalty incurred when A_i and A_j conflict (e.g., an approximation to the time one procedure must wait for the other to release a lock). The *conflict cost* of the lock assignment λ is the sum of the conflicts over pairs of atomic sections sharing some lock:

$$\mathsf{conflict}_\lambda(\mathcal{A}) = \sum_{1 \le i \le j \le n} \mathsf{cost}(i,j) \cdot \delta(\mathsf{locks}_\lambda(A_i) \cap \mathsf{locks}_\lambda(A_j)) \quad (2)$$

where $\delta(S) = 1$ if $S \neq \emptyset$, and $\delta(S) = 0$ otherwise.

The *lock allocation optimization problem* asks to find a lock assignment $\lambda : X \to L$ for a minimal set of locks L such that the conflict cost is minimized. These are conflicting requirements: a singleton lock set and the lock assignment mapping each variable the only lock clearly minimizes the number of locks, but sacrifices a high conflict cost, while a one-to-one map from variables to locks always achieves the optimal conflict cost, but results in the maximum number of locks.

Formally we model the optimization problem as a multi-objective minimization problem. First we notice that the number of locks to can be bounded by $|X|$; we fix a set of locks $L = \{\ell_1, \ldots, \ell_k\}$. For each pair $i, j \in \{1, \ldots, k\}$, we introduce a 0-1 variable a_{ij} which takes the value 1 if and only if global variable x_i is assigned lock ℓ_j. We then represent the number of locks and conflict cost by terms over these variables. The total number of locks used by an assignment is

$$|L| = \sum_{j=1}^{k} (\bigvee_{i=1}^{k} a_{ij}), \quad (3)$$

while the conflict cost $\mathsf{conflict}(\mathbf{a})$ for an assignment is

$$\sum_{1 \le i \le j \le n} \mathsf{cost}(i,j) \cdot \bigvee_{k=1}^{n} \left[\bigvee_{x_l \in A_i} a_{lk} \cdot \bigvee_{x_m \in A_j} a_{mk} \right] \quad (4)$$

(where $\vee$ denotes boolean disjunction). The lock allocation optimization problem is then

$$\begin{array}{ll} & \min \mathsf{conflict}(\mathbf{a}) \\ & \min |L| \\ \text{s.t.} & \sum_{j=1}^{k} a_{ij} = 1 \qquad \text{for each } i \in \{1, \ldots, k\}. \end{array} \quad (5)$$

The constraints maintain that each variable is assigned exactly one lock. A solution of the optimization problem (5) induces a lock assignment; if $a_{ij} = 1$ appears in the solution, then variable x_i is assigned lock l_j. Since each variable is assigned one lock, the induced assignment is a total function. For a program P, we use $\mathsf{Opt}(P)$ to denote a lock assignment induced by an optimal solution to the lock allocation optimization problem 5.

As might be expected, the corresponding decision problem, which attempts to find a lock allocation with less than or equal to k locks, with a conflict cost less than k, is NP-complete.

PROPOSITION 1. *Lock allocation is NP-complete.*

Proof: A nondeterministic polynomial-time algorithm guesses a lock allocation and checks the conflict cost and the number of locks

```
global balance, name;
atomic b1 { access(balance); }
atomic b2 { access(name); }
```

$$\min \mathsf{cost}(b1, b2) \cdot \begin{bmatrix} a(balance, 1) \wedge a(name, 1) \\ \vee \\ a(balance, 2) \wedge a(name, 2) \end{bmatrix}$$

s.t.

$$a(balance, 1) + a(balance, 2) = 1$$
$$a(name, 1) + a(name, 2) = 1$$
$$(a(balance, 1) \vee a(name, 1)) + (a(balance, 2) \vee a(name, 2)) \leq m$$

Figure 1. (a) Source program, (b) Constraints for i locks

are less than the input bounds. We show NP-hardness by reduction from graph coloring. Given a graph $G = (V, E)$ with n nodes and a number of colors k, we produce a program with n global variables $\{x_v \mid v \in V\}$ and n atomic sections $\{A_v \mid v \in V\}$ such that $\mathsf{access}(A_v) = \{x_v\}$ for all $v \in V$. For each edge $(u, v) \in E$, there is a conflict cost $k + 1$ between A_u and A_v, and for each pair $(u, v) \notin E$, there is a conflict cost 0 between A_u and A_v. If the graph can be k-colored, then there is a lock allocation with less than or equal to k locks, such that the conflict cost is less than k. On the other hand, if the graph cannot be k-colored, then there are two atomic sections receiving the same lock and the conflict cost is at least $k + 1$. □

Although the optimization problem (5) is multi-objective and non-linear, a 0-1 integer linear programming (ILP) is achievable with some manipulation. Since the number of locks is bounded by the number of variables, one instead poses k optimization problems $\{p_1, \ldots, p_k\}$ in which only $\mathsf{conflict}(\mathbf{a})$ is minimized, and each p_m has the additional constraint

$$\sum_{j=1}^{k} (\bigvee_{i=1}^{k} a_{ij}) \leq m, \tag{6}$$

stating that the number of locks is less than or equal to m. Second, given the nonlinear objective function and the nonlinear constraint (6), we perform the following general transformation. For each expression of the form $\vee_{a \in A} a$ we introduce a new variable $a_{\vee A}$ to replace it, and add the constraints

$$\textstyle\sum_{a \in A} a \geq a_{\vee A} \quad \text{and} \quad \sum_{a \in A} a \leq |A| \cdot a_{\vee A}. \tag{7}$$

Finally, we replace each monomial $\prod_{a \in A} a$ with the new variable $a_{\prod A}$, adding the constraints

$$\textstyle\sum_{a \in A} a \geq |A| \cdot a_{\prod A} \quad \text{and} \quad \sum_{a \in A} a - a_{\prod A} \leq (|A| - 1). \tag{8}$$

EXAMPLE 1: Figure 1(a) shows a program layout with two global variables `balance` and `name`, and two atomic sections accessing `balance` and `name` respectively—we assume that each atomic section can be executed by concurrently running threads. Intuitively, if a single lock is used, both variables must share it; on the other hand when using two locks it is advantageous to assign each variable its own lock. Figure 1(b) shows problem instance $p_m \in \{p_1, p_2\}$ of the ILP corresponding to the source program in (a). The first two constraints ensure the variables `balance` and `name` each receive exactly one lock, while the third limits the total number of locks to m (where, again, m can be 1 or 2). The boolean disjunctions in the third constraint are removed by adding two variables y_1 and y_2, replacing the third constraint with $y_1 + y_2 \leq m$, and adding the constraints (as required by Equation 7):

$$y_1 \leq a(balance, 1) + a(name, 1) \leq 2y_1$$
$$y_2 \leq a(balance, 2) + a(name, 2) \leq 2y_2.$$

□

Locking Instrumentation. Let λ be a lock assignment to the variables X of the program P. We instrument the program P with `acquire` and `release` statements that take and release locks. We use a *two-phase* locking scheme [13] that acquires all locks before releasing any lock. We use the notation P_λ to refer to the program P instrumented with the acquisitions and releases of locks according to λ. In particular, immediately before each access to a variable $x \in X$ is placed the statement `acquire`$(\lambda(x))$, and upon exit of an atomic section are placed the statements `release`$(\lambda(x))$, for each $x \in X$. The semantics of these statements are respectively to obtain the lock $\lambda(x)$ if it is not yet held by the current thread, and to release $\lambda(x)$ if it is held by the current thread, and the currently executing atomic section is the only one on the call stack. This instrumentation ensures the following.

THEOREM 1. **[Soundness]** *Let P be a multi-threaded program over global integer variables X with atomic sections $\mathcal{A}$, and $\mathsf{Opt}(P)$, an optimal solution to the lock allocation problem for P. For each $A \in \mathcal{A}$, the block A executes atomically in $P_{\mathsf{Opt}(P)}$.*

3. Lock Allocation with Pointers

We now extend the basic technique to programs with pointers.

3.1 Lvalues and Aliasing

Lvalues. A record type is a set of pairs $\langle f, \tau \rangle$ where f is a field name, and τ is the (integer or record) type of f. Records are represented in the standard way: a record value is a reference to a heap structure that stores the values of its fields. An *lvalue* is a term of the form x for some integer or record-valued variable declared in the program, or a field access of the form $v.f$ where v is an lvalue whose record-type contains the field name f. The set of all lvalues which appear in a program's text is denoted Lvals.

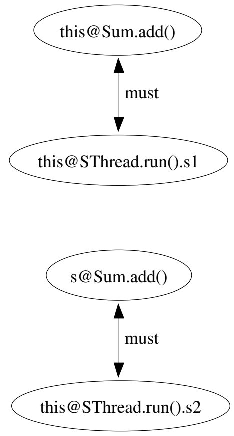
Our analysis generates lock assignments to protect accesses to records, individual fields of records, and global variables. The back-end of our analysis augments each record with a system lock resource which is acquired and released with the associated locking primitives; the acquisition and releasing of locks is also inserted into the target program as described in Section 2. A lock is called *local* if it is a field of a record type, and globally-scoped static locks are called *global*. Local locks protect individual instances of a record or field. On the other hand, if a record or field lvalue is assigned to a global lock, all instances of that record/field are protected by the same lock.

May- and Must-Aliases. For every pair of lvalues v_1 and v_2 of a program P, we say that v_1 and v_2 are *may aliased* if there exists some execution of P such that v_1 and v_2 point to the same heap location at some point along the execution. We say that v_1 and v_2 are *must aliased* if along every execution of P, v_1 and v_2 point to the same heap location, whenever both the lvalues are defined. There are several techniques for statically computing conservative approximations of the may and must alias relations from the text of the program [1].

EXAMPLE 2: **Dynamic Names** In figure 2(a) we have a simple class `Sum` computing a series of additions with the expected values `th1.s1.sum = 300` and `th1.s1.num = 100` upon the exit of `main`. This condition is only guaranteed when the method `add` is executed atomically, as there are otherwise race conditions on the increments of `this.num` and `this.sum`: an interleaving where thread `th1` reads the value of `this.sum`, then thread `th2` reads and updates the value, and finally `th1` updates the value, results in the "lost update" symptom observed from `th1`. □

```
class Sum {
  int num = 0, sum = 0;

  void add(Sum s) atomic {
    this.num += s.num;
    this.sum += s.sum;
    return;
  }
  static class SThread extends Thread {
    Sum s1, s2;
    public void run() {
      int count = 50;
      while (count > 0) {
        s1.add(s2);
        count--;
  } } }
  static void main(String[] args) {
    SThread th1 = new SThread();
    SThread th2 = new SThread();
    th1.s1 = th2.s1 = new Sum();
    th1.s2 = th2.s2 = new Sum();
    th1.s2.num = 1; th1.s2.sum = 3;
    th1.start(); th2.start();
} }
```




```
class Sum {
  int num = 0, sum = 0;

  final Lock lock = new ReentrantLock();

  void add(Sum s) {
    this.lock.lock();
    this.num += s.num;
    this.sum += s.sum;
    this.lock.unlock();
    return;
  }
  ...
}

class Sum' {
  ...
  static void main(String[] args) {
    SThread th1 = new SThread();
    SThread th2 = new SThread();
    th1.s1 = th2.s2 = new Sum();
    th1.s2 = th2.s1 = new Sum();
    th1.s1.num = th1.s1.sum = 1;
    th1.start(); th2.start();
} }
```

Figure 2. (a) Concurrently executing threads in the `Sum` module (b) Must-aliases of `Sum` (c) The rewritten `Sum` program, protected by locks (d) A similar program, `Sum'`, in danger of deadlock

3.2 Constraints

Let $\mathsf{Lvals} = \{v_1, \ldots, v_k\}$ be the set of lvalues of a multi-threaded program P. In our framework, each lvalue v_i has an associated *local lock* resource ℓ_i associated with the runtime object referred to by v_i. In addition, a set $\{g_1, \ldots, g_k\}$ of globally-scoped, static, *global lock* resources are available for allocation, making the entire set of locks $\mathsf{Locks} = \{\ell_1, \ldots, \ell_k\} \uplus \{g_1, \ldots, g_k\}$. Let $\# : \mathsf{Locks} \to \{1, \ldots, 2k\}$ be a one-to-one lock-numbering function. In the same spirit as section 2, we introduce a 0-1 variable $a_{i\#\ell}$ which takes the value 1 if and only if v_i is assigned lock ℓ.

As before, each lvalue must be assigned a lock, thus demands the constraint for each $v_i \in \mathsf{Lvals}$:

$$\sum_j a_{ij} \geq 1 \tag{9}$$

In the face of reference based lock allocation, we maintain the invariant that whenever two lvalues are aliased at run time, their protecting locks are also aliased. We approximate this invariant with the following constraints. For every pair of lvalues v_{i_1} and v_{i_2} which *must* be aliased during program execution (i.e. they are aliases, and are always aliased when accessed), and each lock ℓ, we have the constraint

$$a_{i_1\#\ell} = a_{i_2\#\ell}. \tag{10}$$

In the case that may-aliases v_{i_1} and v_{i_2} may not be aliases at some point of access, we have the constraints

$$a_{i_1\#g} = a_{i_2\#g} \tag{11}$$

for each global lock g, and

$$a_{i_1\#\ell} = a_{i_2\#\ell} = 0 \tag{12}$$

for each local lock ℓ. Constraint (12) conservatively ensures the invariant holds by prohibiting local locks to protect lvalues with indefinite aliases.

EXAMPLE 3: **[Local Locks]** Since the lvalues `this@add` and `s@add` of figure 2(a) have only must-aliases (see figure 2(b)) they can be safely protected by local locks (the application of constraint 12 is avoided). The ILP constraints due to aliasing are thus:

$$a_{\texttt{this@add},\ell} = a_{\texttt{s1},\ell}, \quad a_{\texttt{s@add},\ell} = a_{\texttt{s2},\ell} \qquad \text{for all } \ell \in \mathsf{Locks}.$$

In figure 2(c) we show the program generated due to the locking assignment where `this@add` and `s@add` (and thus `th1.s1`, `th2.s1` and `th1.s2`, `th2.s2`, respectively) are both assigned the instance lock of `this@add`. □

Since locks can be associated with runtime references, we must also ensure that any assigned lock is accessible from every scope in which the respective lvalue is accessed. For each lvalue v_i and each local lock ℓ which is inaccessible from some atomic section $\mathcal{A}$ in which $v_i \in \mathsf{access}(\mathcal{A})$ we have the constraint

$$a_{i\#\ell} = 0. \tag{13}$$

Preferring the assignment of a local lock to a global lock corresponds to reducing the possibility of atomic sections conflicting during program execution. For example, the parallel execution of the same atomic section is possible if only local locks are acquired, and the accessed instance data are not aliased. This preferential treatment can be achieved by giving a higher cost to global locks than local locks, or by making use of specialized features of solvers, such as special ordered sets [3].

THEOREM 2. **[Soundness]** *Let P be a multi-threaded program with atomic sections $\mathcal{A}$, and $\mathsf{Opt}(P)$ an optimal solution to the lock allocation problem for P. For each $A \in \mathcal{A}$, the block A executes atomically in $P_{\mathsf{Opt}(P)}$.*

3.3 Avoiding Deadlock

When the set of variables and locks can be statically determined, as was the case in Section 2, we can avoid deadlock by imposing a linear ordering on the set of locks. In the presence of references and aliasing, the set of locks is determined dynamically. Because locks can be associated with *instances* of program variables (as opposed to a single lock governing every instance of a variable), the target program has the possibility of deadlock, as Example 3.3 demonstrates. We say a program P is *deadlock-free* if every multi-threaded execution of P avoids deadlock.

```
int[] g = { 0, 0 };
void inc(boolean b) atomic {
  if (b) g[0]++; else g[1]++;
}
```

Figure 3. Where alias-precision can make a difference

EXAMPLE 4: **Deadlocks** Figure 2(d) shows another program concurrently performing additions. In this example, the threads `th1` and `th2` are accessing the two `Sum` objects in reversed roles. The lvalues `th1.s1` and `th1.s2` are aliased to `this@add` and `s@add`, respectively, while `th2.s1` and `th2.s2` are aliased to `s@add` and `this@add`. If the competing threads were to each take one lock (the lock of `s@add`, for example) before either took a second lock, the program would deadlock as neither thread can acquire its second lock. Any sound lock assignment must assign the same lock to `s@add` and `this@add` to prevent this possibility. □

To deal with lock ordering in this situation, we analyze the temporal relationships between protected accesses of lvalues.

Accessed-Before Relation. The ordering on locks is obtained via an *accessed before* relation specifying data dependencies on lvalues in the atomic sections. Formally, for two lvalues $v_1, v_2 \in \mathsf{Lvals}$, we say v_1 is accessed before v_2 if there exists an atomic section A and a control-flow path in A such that (1) v_1 is accessed on the path, (2) v_1 is then possibly modified (e.g., by a write to an lvalue that may alias v_1), and finally (3) v_2 is accessed after the modification. We can combine the aliasing information with a traversal of the control-flow graph to obtain a conservative overapproximation of the accessed before relation. If the access graph is acyclic we can order the lvalues of the program linearly by a topological sort of the access graph. If not, we can order the SCC dag of the graph linearly.

Ordering constraints We can enforce a linear ordering (and hence the absence of deadlock) by adding *ordering constraints* to the ILP. Let Deps be an access graph of the program P. We take a linear ordering determined by the scc dag of the dependency graph, and add constraints that state that an lvalue i in a non-trivial scc cannot be assigned local locks for any lvalue in its own scc or in an scc succeeding it in the linear order.

THEOREM 3. *Let P be a multi-threaded program with atomic sections $\mathcal{A}$, and $\mathsf{Opt}(P)$ an optimal solution to the lock allocation problem for P. $P_{\mathsf{Opt}(P)}$ is deadlock-free.*

3.4 Precision

The imprecision of the underlying alias (and dependency) analyses can cause our lock allocation scheme to infer sub-optimal, but still sound, locking assignments. Example 5 illustrates one way this can happen.

EXAMPLE 5: The procedure `inc` of figure 3 simply increments one of two counters, depending the value of its argument `b`. An alias analysis which considers every cell of an array aliased would force our lock allocation scheme to assign the same lock to both cells, despite that it's sound in this case to assign separate locks to each. This allocation causes an unnecessary conflict between callers invoking `inc(true)` and `inc(false)`. □

We have also assumed that any two atomic sections can execute concurrently. Further static analysis to discover which atomic sections may actually execute in parallel [15] would improve the generated locking assignments, since any constraints arising from sections which cannot execute in parallel may be safely pruned away.

4. Experiments

Implementation. We have two implementations of the lock allocation algorithm: `jla` for Java programs and `cla` for C programs. The Java version, `jla`, is coded in Java, using the Polyglot compiler framework [17]. The C version, `cla`, is coded in Ocaml and uses the CIL [16] infrastructure for C programs. Both tools use Minisat+ [4], a pseudo-boolean optimization solver, that converts the 0-1 ILP to a boolean SAT instance. In our experiments, we assigned the following costs to locks and conflicts. Each conflict was given a cost 1. Each global lock was given a cost 2 and each local lock a cost 1. Instead of solving a sequence of optimization problems, we minimized the sum of the conflicts and the sum of the total lock costs. We experimented with a set of different cost heuristics, but the above simple heuristics provided lock assignments that were similar to manually coded locks in most examples.

Nested Atomic Sections. Atomic sections may transitively call methods containing other atomic sections. A lock obtained in a nested atomic section cannot always be safely released at the end of that section—the same lock may be re-obtained by another nested section under the same parent, potentially violating atomicity. To avoid this issue, we use a *two-phase* locking discipline [13]. Locks are obtained when the associated lvalue is first accessed, but are not released until the outermost atomic section finishes. This approach guarantees atomicity and has been used by other locking tools, such as Autolocker [14].

Microbenchmarks. We ran both the C lock allocator and the Java lock allocator on simple string buffer and hash table examples. The string buffer example was a simplification of Java's StringBuffer class. We found local locking assignments that, together with our use of two-phase locking, removed the atomicity violation present in Java's StringBuffer class and reported in [7].

The hash table example, like the `Hashtable` class provided by Java's collection framework, uses an array of linked-list buckets. However, due to methods like `resize` that copy the array, our allocator assigns a single global lock for the hash buckets of `Hashtable`. The `resize` method changes the number of hash buckets by allocating a new bucket array and re-hashing all entries to the new set of buckets, resulting in a cyclic dependency graph. A hand-implemented lock assignment would be better, assigning a lock for each array element. This is left as future work.

AOLServer. We also ran the C lock allocation on a larger program—the `nsd` module of AOLserver [2], an open source webserver in production use at AOL and other companies. The developers of Autolocker had annotated the program with atomic blocks, as well as assigned, to each global variable, a unique lock that protected it. In our experiments, we ignored the lock assignments, and used the atomic annotations to infer a lock mapping and corresponding placement. We ran each file in the `nsd` module separately. This was sound because the global variables were within the scope of the individual files. We test two hypotheses about lock allocation: first, many real programs use very simple locking disciplines that should be automatically inferable; and second, our algorithm performs efficiently in practice.

The results of the AOLserver experiment are summarized in Table 1. The experiments are more a validation of the scalability of the approach rather than to demonstrate particularly intricate locking behavior. In the AOLserver module considered, most files had very simple lock assignments, all of which were equivalent to the assignments synthesized by our lock allocation algorithm. This supports our claim that in many cases, a simple locking scheme is enough to ensure atomicity and this can be automatically inferred.

In most cases, the lock allocation algorithm runs in a few seconds. The constraint generation phase is slow: we have a naive loop that runs in time $O(a^2L^3)$ where a is the number of atomic

File	LOC	Lvals	Atomic	Variables	Constraints	Generate	Solve	Total	Locks	Autolocker
cache	1571	71	10	171551	671068	82.64	0.44	95.93	1	1
callbacks	580	29	7	10883	40860	1.19	0.07	2.56	1	1
dns	522	10	2	388	1189	0.02	0.001	0.53	1	1
driver	1640	26	7	5099	18145	0.390	0.130	15.8	2	2
encoding	835	15	1	241	327	0.01	0.001	1.02	1	1
fd	311	7	2	218	674	0.010	0.001	0.44	1	1
info	734	6	2	157	449	0.001	0.001	0.65	1	1
listen	318	22	3	3915	14152	0.36	0.02	1.07	1	1
log	914	10	1	111	146	0.01	0.01	1.04	1	1
tclenv	296	6	1	43	57	0.001	0.001	0.6	1	1
tclfile	1292	21	5	7338	28142	1.64	0.06	11.63	1	1
tclhttp	578	12	4	749	2528	0.05	0.06	1.71	1	1
tclinit	1434	6	1	43	53	0.001	0.001	2.62	1	1
tclvar	1052	144	11	1350007	5337645	2294.46	23.73	2399.0	4	2
sockcallback	559	13	5	1328	4668	0.09	0.02	1.02	1	1
urlspace	2153	15	6	1291	4589	0.19	0.01	3.96	2	1
unix	568	20	4	3830	14053	0.32	0.02	1.08	1	1

Table 1. Experiments on aolserver. File gives the name of the file in the `nsd` module. LOC is lines of code. Lvals is the number of lvalue names considered by the lock allocation. Atomic is the number of atomic blocks in the code. Variables and constraints give the total number of 0-1 variables and the total number of constraints in the ILP. Generate is time in seconds to generate the ILP from the code. Solve is time in seconds taken by Minisat+ to solve the ILP instance. Total is the wallclock time in seconds for the analysis. Locks is the number of locks inferred by the analysis and Autolocker gives the number of locks provided as Autolocker instrumentation.

blocks and L the number of lvalues. Notice, for example, that the largest example (tclvar) takes almost 40 minutes. However, even though the generated ILP is very large, the constraint solver is extremely fast, taking 24s on this largest example, and usually taking less than 1s. This is because many of the accessibility and aliasing constraints severely restrict the possible assignments (setting many variables to 0) so that boolean constraint propagation in the SAT solver is very effective in pruning the search space.

The lock assignments were similar to those manually specified in the original program except for two cases, tclvar and urlspace. In tclvar, there are two functions with static variables that are changed to global variables by CIL. These variables are assigned locks by our algorithm, but Autolocker did not associate locks with these variables, probably because these would not be modified in parallel. Further, the lock allocation for tclvar finds an instance-specific lock: it associates a lock with each structure of type *Bucket* (which contains a hash table that is protected by this lock). The Autolocker annotations do the same.

In urlspace, there are two global variables `urlspace` and `nextid` that are assigned the same lock by the annotations in Autolocker. As these variables are accessed in separate atomic blocks, our algorithm assigns distinct locks to them. However, our algorithm found an optimal solution that assigned the local lock nominally associated with `urlspace` to `nextid` and vice versa. Thus, the lock assignments found by the solver can be counter-intuitive (though sound).

Acknowledgments. We thank Bill McCloskey for forwarding the Autolocker annotated files to us.

References

[1] L.O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, 1994.

[2] Aolserver. http://www.aolserver.com.

[3] E.M.L. Beale and J.A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In *Proc. Intl. Conf. Oper. Res.*, pages 447–454, 1970. Tavistock.

[4] N. Een and N. Sorensson. Translating pseudo-boolean constraints into SAT. *JSAT*, pages 1–26, 2006.

[5] C. Flanagan and S.N. Freund. Type-based race detection for Java. In *PLDI 00*, pages 219–232. ACM, 2000.

[6] C. Flanagan, S.N. Freund, and M. Lifshin. Type inference for atomicity. In *TLDI 05*, pages 47–58. ACM, 2005.

[7] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI 03*, pages 338–349. ACM, 2003.

[8] J. Gray and A. Reuter. *Transaction processing: Concepts and Techniques*. Morgan-Kaufmann, 1993.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 03*, pages 388–402, 2003.

[10] M. Hicks, J.S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *TRANSACT '06*, 2006.

[11] G. Holzmann. The Spin model checker. *IEEE TSE*, 23(5):279–295, 1997.

[12] S. Jagannathan and J. Vitek. Optimistic concurrency semantics for transactions in coordination languages. In *Intl. Conf. on Coordination Models and Languages*, LNCS 2949, pages 183–198. Springer, 2004.

[13] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Language Design for Reliable Software*, pages 128–137, 1977.

[14] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL 06*, pages 346–358. ACM, 2006.

[15] G. Naumovich, G.S. Avrunin, and L.A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *FSE 99*, pages 338–354. ACM, 1999.

[16] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02*, pages 213–228, 2002.

[17] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An Extensible Compiler Framework for Java. In *CC '03*, pages 138–152, 2003.

[18] M.F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP 05*, pages 92–104. ACM, 2005.

[19] S. Savage, M. Burrows, C.G. Nelson, P. Sobalvarro, and T.A. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 15(4):391–411, 1997.

[20] N. Sterling. Warlock: a static data race analysis tool. In *USENIX Technical Conference*, pages 97–106, 1993.