Inferring Locks for Atomic Sections

Sigmund Cherem *
Department of Computer Science
Cornell University
Ithaca, NY 14853
siggi@cs.cornell.edu

Trishul Chilimbi Sumit Gulwani
Microsoft Research
One Microsoft Way
Redmond, WA 98052
{trishulc, sumitg}@microsoft.com

Abstract

Atomic sections are a recent and popular idiom to support the development of concurrent programs. Updates performed within an atomic section should not be visible to other threads until the atomic section has been executed entirely. Traditionally, atomic sections are supported through the use of optimistic concurrency, either using a transactional memory hardware, or an equivalent software emulation (STM).

This paper explores automatically supporting atomic sections using pessimistic concurrency. We present a system that combines compiler and runtime techniques to automatically transform programs written with atomic sections into programs that only use locking primitives. To minimize contention in the transformed programs, our compiler chooses from several lock granularities, using fine-grain locks whenever it is possible.

This paper formally presents our framework, shows that our compiler is sound (i.e., it protects all shared locations accessed within atomic sections), and reports experimental results.

1. Introduction

One of the main problems when developing concurrent software is to maintain a consistent view of the shared state among all the concurrent threads. For many years programmers have used pessimistic concurrency to develop these applications. Pessimistic concurrency consists of blocking the execution of some threads in order to avoid generating an inconsistent shared state, for instance preventing data races. However, developing reliable and efficient applications with pessimistic concurrency is an arduous task. (a) Programmers need to use fine-grain locks to minimize contention, and ensure that the used locks are sufficient to eliminate the possibility of a harmful data race. (b) Also programmers need to carefully order their operations to avoid dead-locks since pessimistic locking does not lend itself to easy compositionality. Since locks might be present in both library code as well as client code, reasoning about deadlocks requires analyzing the composition of library and client code. (In particular, absence of deadlocks in the library code and in the client code does not guarantee absence of deadlock in the composition of the library and client code). Even after mastering all these challenges, programmers do not always have a guarantee that the written locks yield the intended program semantics.

In recent years, researchers have adopted the concept of *atomic sections* from the Database community as a possible alternative. Atomic sections allow users to give a high level specification of the concurrency semantics. A section of code protected by a keyword *atomic* in some thread must be executed atomically with respect to any other atomic section in any other thread. This is, for any program execution, there exists some other execution where all atomic sections are executed in some serial order, and the final state of both executions is the same. This is commonly referred to as *weak atomicity*. These semantics are enforced by the underlying system without the intervention of the programmer. For programmers this is a very attractive alternative, since all the difficulties of manual pessimistic concurrency are abstracted away.

A natural implementation of atomic sections is to use optimistic concurrency. Via the use of specialized transactional memory hardware (TM) [8, 7], or a software transactional memory (STM) that emulates such hardware [16, 6, 12]. These systems treat atomic sections like transactions and allow them to run concurrently. Whenever a conflict occurs, one of the conflicting transactions is rolled-back and re-executed. Some systems [15, 4] would prevent conflicts using locks, but roll-back transactions when a deadlock occurs. An optimistic system is typically desired when conflicts are rare and hence rollbacks seldom occur.

However, optimistic concurrency has several disadvantages. First, supporting rollbacks and detecting conflicts can impose a considerable amount of runtime overhead. Second, not all atomic sections can be rolled-back, for example after observable actions are performed. A well designed pessimistic approach could avoid all these disadvantages.

This paper presents an automated system to support atomic sections using pessimistic locking. Our system consists of a compiler framework and a runtime library. The compiler performs a source to source transformation that reads programs with atomic sections and produces new programs that use locks to implement such sections. The transformed programs can be run with the use of a special lock runtime library. This approach allows users to run programs written with atomic sections without the need of specialized hardware or an STM.

Our compiler provides several guarantees about the transformed programs. The generated programs must: (a) satisfy the semantics specified by the atomic sections, (b) be deadlock free, and (c) avoid unnecessary thread contention introduced by locks. The last property is necessary to avoid using trivial locking schemes that will induce a large runtime overhead, for example, using a single global lock to protect all atomic sections.

This paper discusses in detail how we generate programs satisfying these properties. First, to satisfy the atomic semantics we formally show that locks introduced by the compiler protect all shared objects used inside an atomic section. Second, to enforce the lack of deadlocks we borrow a locking protocol from the database community. Third, to reduce contention we introduce locks of multiple granularities, and we attempt to use fine-grained locks as much as possible.

The following summarizes our contributions:

- We introduce a set of formal definitions to reason about locks. We define the notion of *abstract lock schemes*, that allows us

* This work was developed while the author was an intern at MSR.

```
 1: void move (list* from, list* to) {
 2:   atomic {
 3:     elem* x = to->head;
 4:
 5:     elem* y = from->head;
 6:     from->head = null;
 7:     if (x == null) {
 8:       to->head = y;
 9:     } else {
10:
11:       while(x->next != null)
12:         x = x->next;
13:       x->next = y;
14:     }
15:   }
16: }
```

(a)

```
acquire(to.head);
elem* x = to->head;
acquire(from.head);
elem* y = from->head;
from->head = null;
if (x == null) {
  to->head = y;
} else {
  acquire(x.next);
  while(x->next != null)
    x = x->next;
  x->next = y;
}
releaseAll();
```

(b)

```
acquireAll({to.head, from.head, R});
elem* x = to->head;

elem* y = from->head;
from->head = null;
if (x == null) {
  to->head = y;
} else {

  while(x->next != null)
    x = x->next;
  x->next = y;
}
releaseAll();
```

(c)

Figure 1. Example program: moves list elements between `from` and `to`. (a) Original program with atomic section. (b) Fine-grain locking scheme susceptible to deadlock. (c) Multi-grain locking scheme avoiding deadlock. `R` is a coarse lock protecting all elements in the `to` list.

to represent locks and the relation between locks of different granularities.

- We present a formal analysis framework to infer locks for an atomic section, given a lock scheme specification as input.
- We show that our analysis is sound. This is, if at the entry of an atomic section each thread acquires the locks inferred by our analysis, then the execution of the program is guaranteed to respect the weak atomicity semantics.
- We present a runtime library required to support our locks of multiple granularities.
- We show experimental results for an instance of our framework. Showing that the analysis scales well to medium size applications, and yields a performance overhead sometimes competitive to an STM approach.

The rest of this document is organized as follows. Section 2 illustrates the ideas behind our system using several examples. Section 3 introduces our formalisms for locks and abstract locks schemes. Section 4 formalizes the analysis framework that infers locks, and discusses how we implemented an instance of this framework. Section 5 discusses a multi-grain locking runtime library used to support the transformations enabled by our analysis. We present our experimental results in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2. Example

This section presents two examples to illustrate the features of our system. The examples use a list data-structure defined in terms of two datatypes:

```
typedef struct elem_t {
    struct elem_t* next; int* data; } elem;
typedef struct list_t { elem* head; } list;
```

Atomic sections with locks Figure 1(a) presents a code that moves elements from one list to another. By the end of the function, the list `from` will be empty, and the list `to` will contain the concatenation of the `to` and `from` lists. The code is wrapped in a atomic keyword, so the entire body of the function should appear to execute atomically. The goal of our system is to introduce locks to enforce the semantics of the atomic section.

A first attempt to write the code using locks would be to acquire a global lock at the entry of the atomic section, and release it by the end of the section. Such approach could introduce a lot of contention: no two threads could run the `move` function concurrently.

A second attempt would be to use fine-grain locks, one lock for each location accessed within the `move` function. Figure 1(b) shows the result of this approach. Before accessing a location we request a lock to protect it. A fine-grain lock that protects a location v is acquired by a call to acquire(e), where e is an expression that evaluates to v. By the end of the atomic section we release all locks. Unfortunately, this code is susceptible to deadlocks. For example, if we run two threads, one calling `move(l1,l2)` and the other calling `move(l2,l1)`. If after the first thread locks `l1.head` on line 2, the second thread locks `l2.head` on the same line, then both threads will be blocked in line 4.

Our system uses a third approach, which consist in avoiding deadlocks by using a locking protocol. The protocol is implemented by acquiring all locks at the entry of the atomic section. To use the protocol, our compiler needs to estimate what locations are accessed within the atomic section, and then introduce locks at the entry of the section to protect each accessed location.

Fine-grain locks can still be used to protect shared locations, as long as the compiler can determine an expression that protects the desired location. However, when atomic sections access unbounded locations, or when there is no expression in scope to protect a shared location, our system introduces coarse-grain locks. Figure 1(c) shows the transformation our system generates for our example. The acquireAll instruction applies the locking protocol on the input set of locks. The locks on `to.head` and `from.head` are fine-grain locks. The lock `R` is a coarse-grain lock used to protect every element of the `to` list.

Finding fine-grain locks Our compiler tries to use fine-grain locks as much as possible. To achieve this goal, the compiler needs to describe the locations accessed in the atomic sections, but in terms of expressions that are in scope by the entry of the atomic block. We use the example in Figure 2 to illustrate how our compiler computes these expressions.

We will focus our attention on the shared location pointed by `z` in line 10. Note that the expression `z` is not in scope at the entry of the section, because `z` is defined later in line 9. The compiler performs a backward tracing to deduce what expressions are equivalent to `z` at the entry of the section. At line 9 the compiler determines that `z` is equivalent to `y->data`. At line 8 the analysis notices that a store assignment on `x->data` is being performed. At this point the compiler needs to carefully consider if the expression

```
 1:   elem* y,x;
 2:   int* w;
 3:   ...
 4:   if (...) {
 5:     x = y;
 6:   }
 7:   atomic {                {y->data, w}
 8:     x->data = w;          {y->data, w}
 9:     int* z = y->data;      {y->data}
10:     *z = null;                {z}
11:   }
```

Figure 2. Analysis example: finding fine-grain protecting locks.

begin traced may be affected by the assignment, which can happen if `x` aliases `y`. In fact `x` will alias `y` if the code in line 5 outside the atomic section is executed. So our compiler must consider both cases: when `x` and `y` are aliases and when they are not. If they were aliases, `x->data` will replace the value of `y->data`, and hence the location pointed by `z` in line 10 would be equivalent to `w` at the entry of the atomic section. In the other case, when `x` and `y` are not aliases, the assigment in line 8 has no effect on `y->data`, and thus `y->data` would point to the location that `z` points to in line 10. Finally, when replacing the atomic section by lock instructions, our compiler acquires both a lock on `y->data` and `w`. This will ensure that the access on line 10 is always protected by some lock.

During this backward tracing, the compiler bounds the size of the expressions it collects. When some expression size exceeds the bound, it is replaced by a coarse-grain lock.

3. Formalizing Locks

This section formalizes our notion of locks. The formalization will be useful to answer questions about locks and atomic sections. For example: What shared locations are protected by a lock? Are two locks protecting a common location? Does a set of locks protect all shared locations accessed in an atomic section?

After introducing our input language, we will proceed by giving a definition of *concrete locks semantics*. We will show how this formal semantics can be used to answer some of the questions above. Then we will instantiate our general semantics definition to give examples of commonly used locks.

In the last part of this section we will introduce the notion of an *abstract lock scheme*. Abstract locks are essentially an approximation of concrete locks that we use to formalize our inference analysis. The formalization will allows us to show that our tool produces programs that respect the atomic semantics.

3.1 Language

Our input language is presented in Figure 3. The language contains standard constructs such as heap allocation, standard assignments and control-flow constructs. The language also includes atomic sections. Expressions used by the statements include variables, dereferences, address-of variables, offsets (indicated by a field offset $i \in F$), allocations, and null values. All values in this language are locations or null, no pointer arithmetic is allowed. Array dereferences and structure dereferences are not distinguished, they are all modeled using field offsets in F. Return statements `return x` are modeled by an assignment $\mathsf{ret}_f = x$, where ret_f is a special variable modeling the return value of f. For simplicity, nested atomic sections are not allowed. Our output language is the same as the input language, except that atomic sections are replaced by two instructions: $\underline{\mathsf{acquireAll}}(L)$, that receives a set of locks L, and releaseAll, that releases all locks held by a thread.

$$
\begin{array}{rcl}
st \in \mathit{Stmt} ::= & x = e \mid *x = e \\
 & \mid & \mathsf{if}(b)\ st\ \mathsf{else}\ st \mid \mathsf{while}(b)\ st \\
 & \mid & st; st \mid \underline{\mathsf{atomic}}\{st\} \\
e \in E ::= & x \mid *x \mid \&x \mid x + i \mid \mathsf{new}(n) \mid \mathsf{null} \\
 & \mid & f(a_0, \ldots, a_n) \\
b \in B ::= & x = y \mid b \vee b \mid b \wedge b \mid \neg b
\end{array}
$$

Figure 3. Input Language

3.2 Concrete Lock Semantics

A lock is simply a name l in some domain LNames that implicitly protects a set of memory locations. We introduce a lock semantics to make this relation between locks and locations more explicit. We write the semantics of a lock l using the denotational function in the domain:

$$[\![\cdot]\!] : \mathsf{LNames} \to 2^{\mathsf{Loc}} \times \mathsf{Eff}$$

where Loc is the domain of memory locations and $\mathsf{Eff} = \{\mathsf{ro}, \mathsf{rw}\}$ is the domain of access effects (reads and writes).

When $(P, \varepsilon) = [\![l]\!]$ we say that l is a lock that, when acquired, protects all locations in the set P, but only to allow the accesses described by ε. For example $(\{v\}, \mathsf{ro}) = [\![l]\!]$ then l ensures that v is protected to be able to read its value, but it is not protected to update its value.

With this definition, we can distinguish fine-grain and coarse-grain locks. A fine-grain lock is a lock that protects a single memory location at all times, formally,

$$\exists v\ .\ [\![l]\!] = (\{v\}, \_)$$

A coarse-grain lock is a lock that may protect more than one memory location.

The domain 2^{Loc} and the subset relation $\subseteq$ form a lattice. We also define a simple two point lattice $(\mathsf{Eff}, \sqsubseteq)$ for the set of effects, where the read-write effect is the top element ($\mathsf{ro} \sqsubseteq \mathsf{rw}$). The domain used in the lock semantics ($2^{\mathsf{Loc}} \times \mathsf{Eff}$) forms a lattice as well, which is defined as the product of the two lattices $(2^{\mathsf{Loc}}, \subseteq)$ and $(\mathsf{Eff}, \sqsubseteq)$. We can use the lock lattice to reason about the relation between locks, for example:

- *Conflict*: two locks conflict if they protect a common location, and at least one of them allows write effects:

$$\begin{array}{l}\mathit{conflict}(l_a, l_b) \Leftrightarrow \\ \quad [\![l_a]\!] \sqcap [\![l_b]\!] \neq (\varnothing, \_) \ \wedge\ [\![l_a]\!] \sqcup [\![l_b]\!] \neq (\_, \mathsf{rw})\end{array}$$

- *Coarser-than*: a lock l_b is coarser-than a lock l_a if it protects all locations protected by l_a, and allows at least the same access effects:

$$\mathit{coarser}(l_b, l_a) \Leftrightarrow [\![l_a]\!] \sqsubseteq [\![l_b]\!]$$

3.2.1 Examples

We now give several example of locks, characterized using our semantics definition.

Expression Locks Program expressions can be used to define fine-grain locks. Let σ denote a program state in our concrete semantics, and consider a program expression e. Whenever the program reaches the state σ, the runtime value of e is always a single location v. This can be written formally using the following relation $\langle \sigma, e \rangle \to v$. To protect v for any read or write access, we can define a fine-grain lock l_e^σ with the following semantics:

$$[\![l_e^\sigma]\!] = (\{v \mid \langle \sigma, e \rangle \to v\}, \mathsf{rw})$$

Global lock A global lock l_g is simply a lock that protects all memory locations:

$$[\![l_g]\!] = (\mathsf{Loc}, \mathsf{rw})$$

Type-based locks In a type-safe language, we could use types to protect all values of such type:

$$[\![l_\tau]\!] = (\{v \mid typeOf(v) = \tau' \wedge \tau' <: \tau\}, \mathsf{rw})$$

where *typeOf* returns the runtime type of a value, and $<:$ is a subtyping relation. In the presence of subtyping, for example with class inheritance in object oriented languages, the super-type is a coarser lock than a sub-type, i.e. $\tau <: \tau' \Rightarrow [\![l_\tau]\!] \sqsubseteq [\![l_{\tau'}]\!]$.

Pointer analysis locks Consider a flow-insensitive and context-insensitive pointer analysis. The analysis abstraction is a set of allocation sites, called *points-to set*. We can define a lock for each points-to set p as follows:

$$[\![l_p]\!] = (\{v \mid allocOf(v) \in p\}, \mathsf{rw})$$

where *allocOf* is a function that returns the site where v was allocated. The lock l_p protects all memory locations allocated in any of the allocation sites in p.

Read and Write Locks A global read lock l_r and a global write lock l_w have the following semantics:

$$[\![l_r]\!] = (\mathsf{Loc}, \mathsf{ro}) \qquad [\![l_w]\!] = (\mathsf{Loc}, \mathsf{rw})$$

Locks Pairs We can also combine the power of two locks sets by computing their Cartesian product. Let l_1 and l_2 be two lock names, we define the concrete pair lock (l_1, l_2) as:

$$[\![(l_1, l_2)]\!] = [\![l_1]\!] \sqcap [\![l_2]\!]$$

This means, the pair lock protects the intersection of the location protected by the individual locks. For example, we can combine expression locks and global read and write locks to obtain a new set of fine-grain locks that protect locations either for read-only or read-write accesses.

3.3 Abstract Lock Schemes

We are now able to reason about locks using their semantics. In this section we explore reasoning about locks in abstract manner, but ensuring that our reasoning is a safe approximation of the concrete locks semantics.

We define an abstract lock scheme Σ as a tuple

$$\Sigma = (\mathcal{L}, \leq, \top, \overline{\cdot}^{\varepsilon}_{p}, +^{\varepsilon}_{p}, *^{\varepsilon}_{p})$$

where elements in $\mathcal{L} \subseteq \mathsf{LNames}$ are lock names, $(\mathcal{L}, \leq, \top)$ is a join-semilattice with top element $\top$. Since $(\mathcal{L}, \leq)$ is a join semilattice, the relation $\leq$ is reflexive, transitive and anti-symmetric. For any pair of elements of $\mathcal{L}$, the join $\sqcup$, that returns their least upper-bound, is defined. For convenience we write $a < b$ to say that $a \leq b \wedge a \neq b$.

The operators domains are the following:

$$\overline{\cdot}^{\varepsilon}_{p} : V \to \mathcal{L} \qquad +^{\varepsilon}_{p} : \mathcal{L} \times F \to \mathcal{L} \qquad *^{\varepsilon}_{p} : \mathcal{L} \to \mathcal{L}$$

where p is a program point in the domain PP and ε is an effect in Eff. The operator $\overline{\cdot}^{\varepsilon}_{p}$ takes a variable symbol and returns a lock name $l = \overline{x}^{\varepsilon}_{p}$; the operations $+^{\varepsilon}_{p}$ and $*^{\varepsilon}_{p}$ are used to relate different locks in $\mathcal{L}$. Together they can be used to express what locations are protected by each abstract lock. We further discuss the semantic meaning of these operators below.

Abstract lock schemes will be used by our lock inference algorithm to compute locks that protect atomic sections. To guarantee that our inference terminates, we require $\mathcal{L}$ to be bounded. Alternatively, we could use widening operators in our formalisms. We decided to make $\mathcal{L}$ bounded to simplify our presentation.

Relation with concrete locks We say that an abstract lock scheme is a *sound* approximation of the concrete semantics, if for any program point p and effect ε, the following conditions are satisfied:

- The top element $\top$ represents a global lock,

$$[\![\top]\!] = (\mathsf{Loc}, \mathsf{rw})$$

- If two locks l_1 and l_2 satisfy $l_1 \leq l_2$, then l_2 must be coarser than l_1:

$$\forall l_1, l_2, \ . \ l_1 \leq l_2 \Rightarrow [\![l_1]\!] \sqsubseteq [\![l_2]\!]$$

- A lock $\overline{x}^{\varepsilon}_{p}$ protects the address of x to be used with the effect ε at the program point p, formally:

$$\forall \sigma, x \ . \ \sigma @ p \wedge \langle \sigma, \&x \rangle \to v \quad \Rightarrow (\{v\}, \varepsilon) \sqsubseteq [\![\overline{x}^{\varepsilon}_{p}]\!]$$

  where $\sigma @ p$ is used to denote that σ is a state that reaches the program point p, and $\langle \sigma, \&x \rangle \to v$ says that the address of x is the location v in the state σ.

- If a location v is protected by l and v' is a location pointed by the field i of v, then $l +^{\varepsilon}_{p} i$ must protect v', formally

$$\forall l \in \mathcal{L}, \sigma, v, v' = v +_{\sigma} i \ . \\ (\{v\}, \mathsf{ro}) \sqsubseteq [\![l]\!] \quad \Rightarrow \quad (\{v'\}, \varepsilon) \sqsubseteq [\![\ l +^{\varepsilon}_{p} i \ ]\!]$$

  where $+_{\sigma}$ performs an offset operation in the concrete semantics of the state σ.

- The operation $*^{\varepsilon}_{p}$ satisfies a similar condition:

$$\forall l \in \mathcal{L}, \sigma, v, v' = *_{\sigma} v \ . \\ (\{v\}, \mathsf{ro}) \sqsubseteq [\![l]\!] \quad \Rightarrow \quad (\{v'\}, \varepsilon) \sqsubseteq [\![\ *^{\varepsilon}_{p} \, l \ ]\!]$$

  where $*_{\sigma}$ performs a value dereference in the concrete state σ.

The combination of $\overline{\cdot}^{\varepsilon}_{p}$, $+^{\varepsilon}_{p}$ and $*^{\varepsilon}_{p}$ allows us to inductively construct a lock that protects the value of any expression e at a program point p to perform an access ε. Let $\widehat{e}^{\,\varepsilon}_{p}$ be such lock, then:

$$\widehat{x}^{\,\varepsilon}_{p} = \overline{x}^{\varepsilon}_{p} \qquad \widehat{e + i}^{\,\varepsilon}_{p} = \widehat{e}^{\,\mathsf{ro}}_{p} +^{\varepsilon}_{p} i \qquad \widehat{*e}^{\,\varepsilon}_{p} = *^{\varepsilon}_{p} \, \widehat{e}^{\,\mathsf{ro}}_{p}$$

Notice that all subexpressions of e only need to be protected for read effects (ro).

3.3.1 Examples

The following are examples of abstract lock schemes, similar to the examples of concrete locks that we gave in the previous section.

Expression locks with k-limiting Expression locks as presented so far can't be used in an abstract lock scheme because the set of locks is not bounded. We introduce k-limiting to bound the set of expression locks, and define a scheme Σ_k as follows:

$$\begin{array}{lcl} \mathcal{L} & = & \{l^p_e \mid length(e) \leq k \wedge p \in \mathsf{PP}\} \cup \{\top\} \\ \leq & = & \{(l^p_e, l^p_e), (l^p_e, \top) \mid l^p_e \in \mathcal{L}\} \\ \overline{x}^{\varepsilon}_{p} & = & \begin{cases} l^p_{\&x} & \text{if } k \geq 1 \\ \top & \text{if } k = 0 \end{cases} \\ l^{p'}_e +^{\varepsilon}_{p} i & = & \begin{cases} l^p_{e+i} & length(e+i) \leq k \wedge p = p' \\ \top & \text{otherwise} \end{cases} \\ *^{\varepsilon}_{p} \, l^{p'}_e & = & \begin{cases} l^p_{*e} & length(*e) \leq k \wedge p = p' \\ \top & \text{otherwise} \end{cases} \end{array}$$

The scheme allows to construct locks l_e for any expression of length k or less. All longer expressions are represented by the $\top$ element. Note that the effect ε is not used in the definitions, hence all locks protect locations for read-write effects (rw).

Unification-based pointer analysis Consider a flow-insensitive unification-based pointer analysis like Steensgard's [17]. Let A be the points-to sets returned by the analysis, such that: each set $s \in A$ is disjoint from the others; each program expression is associated with a points to set (which we write as $e : s$); and points-to relations are denoted by edges $s \to s'$. A sound abstract lock scheme $\Sigma_{\equiv}$

based on the result of this analysis would be:

$$\begin{array}{lcl} \mathcal{L} & = & \{l_s \mid s \in A\} \cup \{\top\} \\ \leq & = & \{(l_s, l_s), (l_s, \top) \mid l_s \in \mathcal{L}\} \\ \overline{x}^{\varepsilon}_{p} & = & s, \text{ where } \&x : s \\ l_s +^{\varepsilon}_{p} i & = & s \\ *^{\varepsilon}_{p} l_s & = & s', \text{ where } s \rightarrow s' \end{array}$$

Read and Write locks We can define a lock scheme Σ_ε that protects locations by the kind of accesses performed in them:

$$\begin{array}{lcllcl} \mathcal{L} & = & \mathsf{Eff} & \overline{x}^{\varepsilon}_{p} & = & \varepsilon \\ \leq & = & \sqsubseteq & l +^{\varepsilon}_{p} i & = & \varepsilon \\ \top & = & \mathsf{rw} & *^{\varepsilon}_{p} l & = & \varepsilon \end{array}$$

Field based locks We can define a lock scheme Σ_i that protects locations by the offset in which they are accessed, as follows:

$$\begin{array}{lcllcl} \mathcal{L} & = & \{s \mid s \subseteq F\} & \overline{x}^{\varepsilon}_{p} & = & \top \\ \leq & = & \subseteq & l +^{\varepsilon}_{p} i & = & \{i\} \\ \top & = & F & *^{\varepsilon}_{p} l & = & \top \end{array}$$

Cartesian product The Cartesian product $\Sigma_1 \times \Sigma_2$ of two abstract schemes Σ_1 and Σ_2, can be constructed by taking the Cartesian product of the domains and functions:

$$\begin{array}{lcl} \mathcal{L} & = & \mathcal{L}_1 \times \mathcal{L}_2 \\ \leq & = & \{((a,b),(c,d)) \mid a \leq b \wedge c \leq d\} \\ \overline{x}^{\varepsilon}_{p} & = & (\overline{x}^{\varepsilon}_{p1}, \overline{x}^{\varepsilon}_{p2}) \\ (a,b) +^{\varepsilon}_{p} i & = & (a +^{\varepsilon}_{p1} i, b +^{\varepsilon}_{p2} i) \\ *^{\varepsilon}_{p} (a,b) & = & (*^{\varepsilon}_{p1} a, *^{\varepsilon}_{p2} b) \end{array}$$

If two abstract lock schemes are sound approximations of the concrete semantics, so is their Cartesian product.

4. Lock Inference Analysis

This section presents the analysis that deduces a set of locks to protect an atomic section. We first present our formal framework, that will allows us to formally show that the analysis is sound. Then we discuss how we implemented an instance of this framework.

4.1 Analysis Formalization

The analysis receives two external inputs: an abstract lock scheme (Σ) and the results of an alias analysis. The alias analysis is useful to understand the effects of store assignments, as in the example from Figure 2. The results of the alias analysis is given by a relation $mayAlias(e_1, e_2, p)$ that answers whether two expressions e_1 and e_2 may point to the same location at a program point p.

For each program point p inside the atomic section, we will compute a set of lock names $N_p \subseteq \mathcal{L}$. To ensure soundness, the analysis result should satisfy the following property: the set of locks N_p protects all the locations used from the point p and forward until the thread reaches the end of the atomic section. Additionally no lock in N_p is redundant in two ways: (a) For any lock $l \in N_p$, there is some location protected by the lock l that is referenced inside the corresponding atomic section (during some run of the program) under the assumption that all program paths are feasible (since our analysis is path-insensitive). (b) For any pair of locks $l_1, l_2 \in N_p$ neither $l_1 < l_2$ nor $l_2 < l_1$.

We formalize the analysis using a dataflow formulation. The algorithm is a backward dataflow analysis that starting at the end of an atomic section, computes the locks for each point until we reach the entry of the atomic section.

Initialization The analysis starts with an empty set $N_0 = \varnothing$ of locks at the end of the atomic section.

Transfer functions Figure 4 presents the intra-procedural analysis rules. Given a set N of lock names at the program point after a statement st, the analysis computes a new set N' for the program point before st. The new set N' must protect all locations protected by N and all locations accessed directly by the statement. Note the rules are formulated using closure operators which are not meant to be used in an implementation. Section 4.3 discusses how this analysis is implemented in practice.

For any assignment $e_1 = e_2$, the transfer function is:

$$\begin{array}{lcl} transfer(e_1 = e_2, N) & = & \{l' \mid (l,l') \in T_{e_1=e_2} \ \wedge l \in N\} \\ & \cup & G^{\mathsf{rw}}_{e_1} \cup G^{\mathsf{ro}}_{e_2} \end{array}$$

T is the underlying transfer function relation:

$$\begin{array}{lcl} T_{e_1=e_2} & = & closure(S_{e_1=e_2} \cup Id) - closure(Q_{e_1}) \\ closure(S) & = & S \\ & \cup & (\{(l+i, l'+i) \mid (l,l') \in closure(S)\} \\ & \cup & (\{(*l, *l') \mid (l,l') \in closure(S)\} \\ Id & = & \{(\overline{z}, \overline{z}) \mid z \in V\} \end{array}$$

S is a set of core changes induced by the statement:

$$\begin{array}{llll} S_{x=y} & = \{(*\overline{x}, *\overline{y})\} & S_{x=y+i} & = \{(*\overline{x}, *\overline{y}+i)\} \\ S_{x=\&y} & = \{(*\overline{x}, \overline{y})\} & S_{x=*y} & = \{(*\overline{x}, *(*\overline{y}))\} \\ S_{x=\mathsf{new}} = S_{x=\mathsf{null}} & = \{\} & S_{*x=y} & = \{(*(l), *\overline{y}) \mid l \sim *\overline{x}\} \end{array}$$

Q are trivial mappings violated by the statement:

$$Q_x = \{(*\overline{x}, *\overline{x})\} \qquad Q_{*x} = \{(*(*\overline{x}), *(*\overline{x}))\}$$

G are new locks to protect the accesses in the current statement:

$$\begin{array}{lll} G^{\varepsilon}_{*x} = \{*^{\varepsilon}\ \overline{x}^{\mathsf{ro}}, \overline{x}^{\mathsf{ro}}\} & G^{\mathsf{ro}}_{x+i} = \{\overline{x}^{\mathsf{ro}}\} & G^{\mathsf{ro}}_{\&x} = \{\} \\ G^{\varepsilon}_{x} = \{\overline{x}^{\varepsilon}\} & G^{\mathsf{ro}}_{\mathsf{new}} = \{\} & G^{\mathsf{ro}}_{\mathsf{null}} = \{\} \end{array}$$

Figure 4. Transfer functions. Annotations on operators are omitted to simplify the presentation.

The figure omits program point and effect annotations to simplify the presentation. The reader should note that every operation $\overline{\cdot}$, $*$ and $+$ on the first component of a pair corresponds to the program point a after the assignment, i.e., $\overline{\cdot}_a$, $*_a$ and $+_a$; and, every operation on the second component corresponds to the program point b before the assignment. However, both use the same effects ε. For example, the relation $S_{x=y}$ must be read as:

$$S_{x=y} = \{(*^{\varepsilon_1}_a\ \overline{x}^{\varepsilon_2}_a, *^{\varepsilon_1}_b\ \overline{y}^{\varepsilon_2}_b)\}$$

For any assignment $e_1 = e_2$ we describe the transfer function as a combination of two basic relations $S_{e_1=e_2}$ and Q_{e_1}. The relation $S_{e_1=e_2}$ includes a minimal set of locks that are changed by the statement. For example, in $S_{x=y}$ any lock that protects $*\overline{x}$ after the statement, is protected by $*\overline{y}$ before the statement. We express how other locks are transformed using the closure operator $closure(S)$. For instance, the transfer function maps $*(*\overline{x}+i)$ after the statement to $*(*\overline{y}+i)$ before the statement.

The closure of Id allows us to express that any expression not affected by the assignment will remain unchanged by the transfer function. The closure of Q_{e_1} are those expressions in $closure(Id)$ that are affected by the statement, and thus must be excluded from the transfer function. For example, the transfer function of $T_{x=y}$ maps $*(*\overline{z})$ to $*(*\overline{z})$, but $*(*\overline{x})$ is not mapped to $*(*\overline{x})$ because $(*(*\overline{x}), *(*\overline{x})) \in closure(Q_x)$.

Finally, all transfer functions include a new set of locks G that protect the locations accessed by the assignment.

The transfer function of $*x = y$ uses a may alias relation $\sim$. We construct $\sim$ from the results of a pointer analysis, which was given as input to this algorithm. If two expressions e_1 and e_2 may alias at a program point p, written $mayAlias(e_1, e_2, p)$. Then the relation $\sim$ holds on their abstract locks, i.e. $\widehat{e_1}^{\,p} \sim \widehat{e_2}^{\,p}$.

We define the transfer of $x = f(a_0, ..., a_n)$ as follows:

$$transfer(x = f(a_0, ..., a_n), N) = \\ trans^*(p_0 = a_0; ...; p_n = a_n; st_{\mathsf{f-body}}; x = \mathsf{ret}_m, N)$$

where $st_{\mathsf{f-body}}$ is the body of f, and $trans^*$ is defined as the least solution to the following recursive formulation:

$$trans^*(st, N) = \\ \begin{cases} trans^*(st_1, N) \sqcup trans^*(st_2, N) & st \equiv \mathsf{if}(b)\ st_1\ \mathsf{else}\ st_2 \\ trans^*(st_1, trans^*(st_2, N)) & st \equiv st_1; st_2 \\ N \sqcup trans^*(st; \mathsf{while}(b)\ st, N) & st \equiv \mathsf{while}(b)\ st, N \\ transfer(st, N) & \text{otherwise} \end{cases}$$

Figure 5. Inter-procedural equations

To illustrate how this formalization works, consider our earlier example from Figure 2. The program's atomic section can be rewritten in our simplified language as follows:

```
atomic {
   t1 = x + data; *t1 = w;
   t2 = y + data; z = *t2;
   *z = null;
}
```

Initially, our analysis starts with an empty set of locks at the end of the section. The transfer function of `*z = null` uses G to introduce two new locks $*\overline{z}$ and $\overline{z}$. Lets focus on what happens to $*\overline{z}$ only. The previous statement `z = *t2` defines z in terms of t_2, our transfer function uses the relation $S_{z=*t_2}$ to transform $*\overline{z}$ into $**\overline{t_2}$. Notice that the pair $(*\overline{z}, *\overline{z})$ is mentioned in the set Q, and thus $*\overline{z}$ is no longer included in the set of locks. Similarly, the transfer function for `t2 = y + data` transforms $**\overline{t_2}$ into $*(*\overline{y} + data)$, which we wrote as `y->data` in Figure 2. The assignment `*t1 = w` requires us to look at the may alias information. Assume that the may analysis indicates that $mayAlias(t_1, y + data, \cdot)$ holds at that point, then the following relation also holds $*\overline{t_1} \sim *\overline{y} + data$. This enables the transformer $S_{*t_1=w}$ to introduce $*\overline{w}$ in the set of locks. Additionally, $*(*\overline{y} + data)$ remains in the set of locks because it is in $closure(Id)$ and it is not listed in $closure(Q)$. Finally, the first assignment `t1 = x + data` doesn't remove any of our locks and we conclude that to protect the access in the last line, we need both $*(*\overline{y} + data)$ and $*\overline{w}$.

Merge operation At merge points we compute the join of two set of locks N_1 and N_2 as follows:

$$N_1 \sqcup N_2 = \{l \in N_1 \cup N_2 \mid \nexists l' \in N_1 \cup N_2\ .\ l < l'\}$$

all locks are combined together, but we exclude locks protecting locations that are already protected by other locks in the set.

Function calls Figure 5 formulates how to reason about function calls. Essentially we model function calls as a composition of three group of statements: a first group assigns actual arguments to the formal arguments used in the callee, a second group is the body of the callee, and a final group consist of assigning the return value of the callee to the left hand side of the call. The formulation uses $trans^*$ to define the transfer function for compound statements. These are essentially summaries of the transfer functions of several statements. As mentioned earlier, these rules are meant as formal declaration of our system, not as an implementation.

Transformation From the analysis solution we retrieve the set of locks N inferred for the entry point of each atomic section. We replace each atomic section with two statements: a statement acquireAll(N) at the entry point of the atomic section, and a statement releaseAll at the end of the atomic section.

4.2 Soundness

To ensure that our algorithm is correct, we need to connect our abstract domain with the program semantics. Our operational semantics consist of states σ, that contain a shared heap and a set of locks L_i held by each thread i. Additionally, our semantics keeps track of the state of each thread, whether it is inside or outside an atomic section. Using our concrete locks semantics $[\![\cdot]\!]$, our operational semantics check that any shared location accessed inside an atomic section is protected by a lock in L_i. In particular, a step in the semantics $\langle \sigma, st \rangle \rightarrow \sigma'$ will get stuck, if the check doesn't hold.

THEOREM 1. *Let σ be a reachable state, where thread i is about to execute a statement st within an atomic section. Let N be the result of our analysis for such atomic section. If at the entry of the atomic section, the thread i acquired all locks in N, then there exists some σ' such that $\langle \sigma, st \rangle \rightarrow \sigma'$, i.e. the program doesn't get stuck.*

Proof. Our proof is based on the assumption that both, the abstract lock scheme and the *mayAlias* query, are sound. The theorem proof is based on induction on the program structure, and it uses the lemmas below to show each step of the proof.

LEMMA 1. *The locks before any assignment protect the locations accessed directly by the statement. Formally,*

$$\forall \sigma, st\ .\ \sigma@(\bullet st) \Rightarrow \exists l \in transfer(*e_1 = e_2, N)\ . \\ \langle \sigma, e_1 \rangle \rightarrow v \Rightarrow (\{v\}, \mathsf{rw}) \sqsubseteq [\![l]\!] \wedge \\ \forall (*e) \in subs(e_i)\ .\ \langle \sigma, e \rangle \rightarrow v \Rightarrow (\{v\}, \mathsf{ro}) \sqsubseteq [\![l]\!]$$

where $\sigma@(\bullet st)$ is any state that reaches the program point before st, $subs(e_i)$ returns all dereference subexpressions of e_1 and e_2, e_1 ranges over $\{\&x, *\&x\}$.

LEMMA 2. *Assume N is a set of locks protecting all locations accessed after a statement st. Except for unreachable locations, the locks inferred by $transfer(st, N)$ protect all the locations that were protected by N after the statement:*

$$\forall st, l \in N, \sigma, v \in \mathsf{Loc}\ . \\ \sigma@(\bullet st) \wedge reach(\sigma, v) \wedge (\{v\}, \varepsilon) \sqsubseteq [\![l]\!] \Rightarrow \\ \exists l' \in transfer(st, N)\ .\ (\{v\}, \varepsilon) \sqsubseteq [\![l']\!]$$

where $reach(\sigma, v)$ holds if the location v is reachable from some program variables in state σ.

Due to lack of space our semantic rules and proofs are omitted here, they can be found in a companion technical report [3].

4.3 Implementation

Our framework is parameterized on an abstract lock scheme, and a pointer analysis. We implemented an instance of this analysis framework for a fixed lock scheme and pointer analysis. Our formulation uses constructs that are can't be implemented in practice, for example the closure operator, and the $trans^*$ function. We now describe how to implement the rules described in our formalisms.

We chose an abstract lock scheme consisting of k-limited expressions, points-to sets and read/write effects ($\Sigma_k \times \Sigma_{\equiv} \times \Sigma_{\varepsilon}$). We use Steensgard's pointer analysis [17] to compute both, the points-to sets in $\Sigma_{\equiv}$, and the *mayAlias* relation.

Our implementation keeps a set of locks for each program point, and uses standard worklist algorithm to compute solutions to the dataflow equations. The algorithm operates at the level of individual locks, not at the level of set of locks that is used in the formalisms. The worklist contains pairs of locks and program points (l, p). Given an atomic section, we initialize the worklist by inspecting each assignment $e_1 = e_2$ within the atomic section, and adding a pair (l, p), if $l \in G^{\mathsf{rw}}_{e_1} \cup G^{\mathsf{ro}}_{e_2}$ and p is the program point before $e_1 = e_2$. We only omit a lock $l = \overline{x}$ if we can tell that x is a thread local variable, whose address is never stored.

When an element (l, p) is taken from the worklist, the statement st before p is analyzed. We use the transfer function for st to compute the effects of the statement on the lock l, without explicitly computing the closure operations used in our formalisms. The result of the transfer function is a set of locks $l'_1, ..., l'_n$, that we merge at the program point p' before st. If the set of locks at p' changes, the pairs (l'_i, p') and are added back to the worklist. This process is repeated until the worklist is empty.

The analysis implementation is specialized to take advantage of the abstract lock scheme we have chosen. In particular, we noticed that from all possible pairs of expressions and points-to set locks, only few combinations need to be manipulated by the analysis. For example, if an expression e belongs to a points-to set P, then the analysis will never consider a pair of e with $P' \neq P$. This is because e and P' protect disjoint sets of locations, and thus their combination protects no memory location. In fact, the relevant pairs of expressions and points-to sets implicitly define a tree hierarchy, instead of a general lattice. This tree consist of a root node $(\top, \top)$ that protects all locations. The root's immediate children are points-to set locks $(\top, P)$ that protect a partition of the memory. Finally, each points-to set has k-limited expressions locks (e, P) as children. These children protect a location within the memory partition protected by $(\top, P)$.

The backward analysis traces these pairs (e, P) through the program, updating the pair lock on each statement. In practice, only the first component e is updated by the analysis. The transfer functions will always conclude that P remains unchanged. This is because P is a flow-insensitive lock, and thus the same lock protects the same set of locations before and after each statement. Once an expression reaches the k-limit and becomes $\top$ the analysis will never change the first component either. Therefore, we exploit these observations in our implementation: our tool only tracks k-limited expressions until they become $\top$, at which point the tracing is stopped, and the corresponding points-to set lock is added to the analysis solution.

We analyze function calls using a standard technique based on *function summaries* [14]. A function summary f_s essentially summarizes the analysis results for the body of a function f. Given a lock l at the end of the function f, $f_s(l)$ is the set of locks that protect the same locations as l at the beginning of the function f. Summaries are similar to the function $trans^*$ in our formalisms, but they are only computed for function blocks.

When analyzing a lock l after a function call $x = f(a_1, .., a_n)$, we perform the following steps:

- First, we *map* the lock to the context of the callee by modeling the assignment $x = \mathsf{ret}_f$. The assignment simulates returning from the call to f and setting the return value to x. Let l_1 be a lock resulting of analyzing such assignment.
- Second, we check if we have a summary for l_1 in f_s. If there is no summary defined, we add (l_1, exit_f) to the worklist, where exit_f is the program point by the end of the function f. If a summary for l_1 is found, then let l_2 be one such lock in the result of the summary $(l_2 \in f_s(l_1))$.
- Third, we *unmap* the lock l_2 back to the context of the caller by modeling how actual arguments are passed as formals to the callee, i.e. $p_i = a_i$. To handle recursive calls, we add *shadow* variables to avoid naming conflicts between the actuals and the formals of the call. Let l_3 be a resulting lock from these assignments, then we merge l_3 at the point p before the call and add (l_3, p) in the worklist if the set of locks has changed.

Additionally, to build function summaries, the analysis keeps track of the origin of a lock l, denoted by $src(l)$. For example, if $l' \in transfer(st, l)$, then they have the same source $src(l) = src(l')$. Sources are initialized at the exit point of a function. When the algorithm reaches the entry point of the function (l, entry_f), the analysis updates the function summary by adding l to $f_s(src(l))$. After updating the summary, the analysis also *unmaps* the lock l to all callers of the current function.

	X	S
X		
S		✓

(a)

	X	I_s	SI_x	S	I_x
X					
I_s		✓	✓	✓	✓
SI_x		✓			
S		✓		✓	
I_x		✓			✓

(b)

Figure 6. Compatibility of access modes: (a) traditional modes, (b) with intention modes.

5. Runtime System

We run the transformed programs using a library that supports the locks generated by our analysis. This section describes how the runtime library is designed and discusses how our transformation is integrated with this library.

5.1 Multi-granularity Locking Library

Our locking schemes define a structure of multi-granularity locks. Unlike traditional locks, there are pairs of locks that cannot be held concurrently. For this reason, using a simple linear order of the locks does not guarantee that locking is deadlock free.

To support multi-grain locks, we implemented a library based on ideas introduced by Gray et al. [11, 10] from the database community. To illustrate the key ideas of a multi-grain locking protocol, consider the following example. Suppose we have a simple lock structure of three locks l_a, l_b, and $\top$, where $l_a \leq \top$ and $l_b \leq \top$. We would like to allow l_a and l_b to be held concurrently. But if a thread acquires $\top$, no other thread can get l_a or l_b. Suppose a first thread wants to acquire l_a. The protocol must ensure that before requesting l_a, $\top$ is not taken by any other thread. One way to do this is to acquire $\top$ and then l_a, but this will not allow another thread to request l_b. Instead, a multi-grain protocol marks $\top$ with an *intention*. This *intention* says that somewhere lower in the structure, this thread holds a lock. When some other thread wishes to acquire $\top$, it must wait until the *intention* mark is removed. However, intention marks are compatible, hence a second thread can also mark $\top$ with his intention, and acquire l_b concurrently with l_a.

More generally, a protocol for multi-grain locks operates based on three basic ideas: (a) lock relationships are structured, (b) locks are requested in a top-down fashion, and (c) dependences between locks are made explicit during the protocol using *intention modes*.

Traditionally, locks can be acquired in two modes: read-only or shared (S), and read-write or exclusive (X). Adding intentions introduces three new modes: intention to read (I_s), intention to write (I_x), and read-only with the intention to write some children nodes (SI_x). Figure 6 shows the compatibility between these access modes. A pair of access modes marked with ✓ can be held concurrently.

If the lock structure is a tree, the following deadlock free protocol guarantees that no conflicting locks are held concurrently:

- Before acquiring l for reads (S) or intention to read (I_s), all ancestors l' ($l \leq l'$) must be held by this thread in I_x or I_s.
- Before acquiring l for X, SI_x or I_x all ancestor l' ($l \leq l'$) must be held by this thread in SI_x or I_x mode.
- Locks are released bottom up or at the end of the transaction.

This protocol can be extended to deal with general lattice structures (not only trees), but we omit this for simplicity. Since our implementation uses a locking scheme that has a tree-like structure, the protocol presented here is sufficient.

5.2 Lock Runtime API

We implemented the multi-grain protocol in a runtime library. Our runtime library API consists of three functions: *to-acquire*, *acquire-all*, *release-all*. The function *to-acquire* takes a *lock descriptor* and adds it to a list of pending locks. We define lock descriptors further below. The function *acquire-all* proceeds to request all pending locks using the protocol presented above, and moves the locks from the pending list to a list of granted locks. Finally *release-all* unlocks all the locks in the granted list and clears the list.

In order to acquire locks using the protocol, our library requires knowledge of the lock structure: for every lock l the protocol accesses all locks in the path from the root $\top$ to the lock l. We provide this structure information to the runtime library in the lock descriptors. A lock descriptor for a lock l is the path $p = (\top, l_1, ..., l_n, l)$ to reach l from $\top$. Thus, we avoid storing the entire lock structure in the runtime library, but instead, we provide the library with the relevant portion of the locking structure using the lock descriptors.

The transformation we presented in Section 4 inserts statements $\underline{\mathsf{acquireAll}}(N)$ for a set of locks $N = \{l_1, ..., l_n\}$, and $\underline{\mathsf{releaseAll}}$ to release all locks. To integrate our transformed programs with our runtime library, our implementation translates a $\underline{\mathsf{releaseAll}}$ statement into a call to *release-all*, and an $\underline{\mathsf{acquireAll}}(N)$ statement to the sequence *to-acquire*(p_1); ...; *to-acquire*(p_n); *acquire-all()*; where p_i is the lock descriptor of a lock l_i.

6. Results

This section describes our experimental results. We first discuss our experiment setup. Then, we present an evaluation of our compiler. Finally, we show runtime statistics of the transformed programs.

6.1 Experiment Setup

Our implementation is divided in three phases. A first phase reads C/C++ programs and outputs each function in a simplified intermediate representation (IR). A second phase reads the IR and performs the whole program analysis as described in Section 4.3. Finally, a third phase compiles the C/C++ programs from scratch, using the results of the previous phase to instrument the programs. All phases are implemented in C#, the first and third phase use the Phoenix compiler infrastructure [1] as a frond and back end. We used several values for k, between 0 and 9, to build the k-limited expressions.

Testing environment We performed all experiments on a 1.66Ghz Dual Core, 2 GB RAM machine, running Windows XP SP2.

We used several benchmarks to evaluate our tool, including programs from the SPECint2000 benchmarks [18], the STAMP benchmarks [2] and a *hashtable* micro-benchmark we wrote. The STAMP and *hashtable* programs are concurrent applications that contain atomic sections protecting shared memory accesses. The *hashtable* program is designed to perform several hashtable operations (*put*, *get* and *remove*). Each operation is enclosed in an atomic section. Each atomic section contains a loop with additional *nop* instructions to make the program spend more time inside the atomic sections. The SPECint2000 programs are not concurrent, but they were used to measure the scalability of the analysis. We wrapped the main function of these programs inside an atomic section, and analyzed them in the same fashion as the concurrent programs.

We used the TL2 software transactional memory [4], distributed with the STAMP benchmarks, to compare the runtime performance of our approach against an optimistic alternative. To make our comparison fair, we used the same compiler to build both, the programs with TL2, and the transformed programs with our multi-grain locking runtime library. The TL2 STM can be compiled under Windows using Cygwin and gcc-3.4.4. We took the results of our analysis and manually extended the benchmarks in order to compile them with *gcc*. Our manual intervention is minimal, it consists of adding calls to our multi-granularity locking library as they would be generated by our compiler.

Program	KLOC	Num atomic sections	Analysis Time (s) $k=0$	Analysis Time (s) $k=9$	Total Time(s) $k=9$
SPEC					
mcf	2.8	1	1.0	1.2	110.7
bzip2	4.6	1	1.5	4.2	44.7
gzip	10.3	1	1.6	3.4	164.2
parser	14.2	1	4.2	11.0	178.8
vpr	20.4	1	5.0	32.6	872.7
crafty	21.2	1	5.3	111.3	636.8
twolf	23.1	1	6.2	15.4	951.4
gap	71.4	1	3.0	76.6	908.0
vortex	71.5	1	10.6	193.7	806.6
STAMP					
vacation	14.1	1	1.1	1.5	175.6
genome	13.1	5	1.5	1.6	153.4
kmeans	11.9	3	0.9	0.9	108.2
hashtable	0.4	4	0.7	0.7	23.6

Table 1. Program size and analysis time in seconds. The analysis takes less than a fifth of the total compilation time.

6.2 Compiler Statistics

Table 1 shows the size of each program analyzed, the analysis time, and the total compilation time. The analysis time includes the time for the unification-based points-to analysis and the backward dataflow analysis. As mentioned earlier in Section 4.3, the dataflow analysis is only performed for expression locks until they become $\top$. Thus, the analysis with $k = 0$ doesn't perform any dataflow computation, so the column of analysis time with $k = 0$ can be used as a rough estimate of the time spent in the pointer analysis. The total compilation time includes the first and third phase of our compiler: parsing the source files, generating the simplified intermediate representation, compiling to native code, and any other phase performed by the Phoenix infrastructure.

The time spent by the dataflow analysis depends on the size of the atomic sections, and the number of shared memory accesses within the atomic sections. Only in the SPEC benchmarks the size of the atomic sections, and therefore the analysis time, is correlated with the program size. For this reason, the SPEC benchmarks use more analysis time than the other programs. Both the STAMP benchmarks and *hashtable* contain small atomic sections; thus the analysis cost is fairly low. The numbers observed are quite promising; they show that our technique can scale to analyze large atomic sections of up to 80 KLOC. The analysis time is less than a fifth of the total compilation time.

Lock Distribution For each value of k we counted the number of locks chosen by our analysis to protect each atomic section. We divide the locks in four categories: (a) expressions or fine-grain read-only locks, (b) fine-grain read-write locks, (c) points-to set or coarse-grain read-only locks, (d) and coarse-grain read-write locks. Figure 7 shows the overall results. Each column shows the combined total number of locks in each category from all atomic sections of every program.

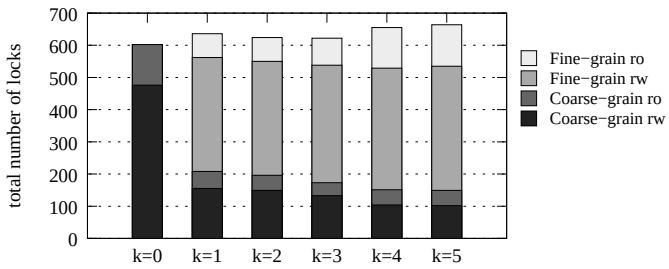

Figure 7. Combined total number of fine-grain and coarse-grain locks from all programs. Increasing the analysis precision reduces the number of coarse locks and possibly reduces contention.

As expected, all locks chosen by the analysis with $k = 0$ are coarse-grain locks. As we increase the value of k we observe that coarse-grain locks can be replaced by one or more fine-grain locks, and sometimes coarse-locks can be removed altogether. The former is illustrated in the column of $k = 1$, where individual coarse-grain locks are replaced by several fine-grain locks (thus the increase in the number of locks). The latter is illustrated when $k = 3$, where the total number of locks is reduced. We expect this decrease is due to objects allocated within the atomic sections: these objects are not reachable by the entry of the atomic section, and thus, they are not shared unless they become explicitly stored in another location. Locks protecting the other location can be used to implicitly protect the allocated cells. The dataflow analysis deduces this when tracing fine-grain locks up to their allocation site.

Beyond $k = 5$ there is no apparent benefit of increasing the value of k. This is because our k-limited locks are used to protect locations in non-recursive structures, for example in globals, structure fields or array entries. Non-recursive structures have a bounded depth, and typically programmers use a depth of 2 or 3 heap dereferences. Recall that both offset operations and heap dereferences contribute to the length of an expression, thus many expressions with 3 heap dereferences may have length $k = 5$.

6.3 Runtime Statistics

We now focus on the STAMP benchmarks and *hashtable* to evaluate the runtime performance of the transformed programs. For most programs we experimented with two parameters configurations: *high* and *low*. The *high* setting is designed to produce more conflicts between transactions, for example, performing more put operations than get operations in *hashtable*. The *low* setting introduces less contention by performing more read-only operations.

Table 2 shows the running time of the evaluated benchmarks. The first column shows the running time using a single thread. The next three columns show the running time when using 2 threads. We didn't collect the running time with more than 2 threads since we evaluated our tool on a 2-core machine. The second column contains the running time when using the TL2 STM to support atomic sections. The last two columns show the time consumed by the transformed applications, when using the locks chosen by the analysis with $k = 0$ and with $k = 9$.

The entries of TL2 for the program *vacation* often failed validation checks when running under TL2. We haven't determined the reason for these violations. The checks were always satisfied by our transformed programs. The reported time for TL2 are instances when the checks didn't fail.

Runtime impact of multi-granularity locks. Our compiler chose coarse-grain locks for most atomic sections in the STAMP programs, except for one atomic section in *genome*. This explains that the running time is very similar for $k = 0$ and $k = 9$. The application *genome* spends most of its time in atomic sections where our analysis uses coarse-grain locks. The fine-grain locks added with $k = 9$ protect short lived transactions, and hence introduce more runtime overhead than coarse-grain locks.

Program	Running time (s)			
	Sequential	TL2	$k = 0$	$k = 9$
genome	5.41	16.77	29.87	35.87
vacation-high	0.29	10.97*	0.35	0.36
vacation-low	0.46	11.28*	0.52	0.51
kmeans-high	2.96	14.64	31.38	33.11
kmeans-low	4.72	12.58	26.45	27.05
hashtable-high	23.88	14.66	26.85	18.08
hashtable-low	23.84	14.77	17.77	17.73

Table 2. Runtime statistics. Entries marked with * failed validation checks on some executions.

The opposite situation was observed in *hashtable*. In this program, the *put* operation only updates a single bucket in the table. The analysis with $k = 9$ assigns *put* operations a single fine-grain lock to protect that bucket entry. In our experiments, the *high* configuration makes *put* operations 8 times more frequent than *get* and *remove* operations. Thus, using fine-grain locks makes the program run 1.48 times faster than using coarse-grain locks ($k = 0$). The *low* configuration makes *get* operations 8 times more frequent than the others. In this case, fine-grain locks bring no benefit over the coarse grain locks.

In *hashtable* we can also observe the benefits of tracking read and write effects. The analysis determined that *get* only performs memory reads. Hence, multiple *get* operations can run concurrently. Since the *low* configuration introduces more *get* operations, the transformation with coarse-locks runs 1.51 times faster in the *low* setting than in the *high* setting.

Comparison with TL2 On average the TL2 system ran 35% faster than the coarse-grain lock alternative ($k = 0$), and 31% faster than the fine-grain lock alternative ($k = 9$). However, the numbers fluctuate a lot across the evaluated benchmarks. In *genome* and *kmeans* the TL2 system ran approximately twice as fast as any of our locking solutions. The reason for this is that our analysis was unable to find additional parallelism from fine-grain locks. This imprecision of our analysis can be improved by choosing a different locking scheme, for example one based on a more precise pointer analysis technique. Additionally, our locking library implementation, which is not optimized, introduces additional runtime overhead.

The *hashtable* program ran 1.2 times faster using TL2 than using fine-grain locks. The additional speed comes from parallelism between *put* and *get* operations. Our technique was able to enable parallelism between pairs of *put* operations (using fine-grain locks) and between pairs of *get* operations (using coarse-grain read-only locks), but *put* and *get* still have contention with each other.

In *vacation*, on the other hand, our locking approach was up to 30 times faster. We noticed that this happens because conflicts are very common in this application. Most transactions in *vacation* access a common global structure, and thus, many transactions are aborted. We instrumented the programs and observed that each atomic section was being executed 4 times on average using TL2.

Final Comments The experimental results show that the analysis scales well, and the transformed programs run at most a factor of 2 slower than the TL2 system. Yet, there is a large room for improvement. For example, our transformed version of *kmeans*, using two threads ran 12 times slower than the sequential version of the program. The slowdown we have observed can be attributed mainly to two issues: most atomic sections are protected by coarse-grain locks, and our runtime system is not optimized. The former can be attributed in part to our analysis, and in part to the non-

parallelizable nature of the benchmark. For some applications, like the STAMP benchmarks, our analysis cannot deduce a set of fine-grain locks to protect the atomic sections. For other applications, like *hashtable* or some of the SPEC benchmarks, the analysis gives better results. We need a better set of benchmarks to measure the benefits of our approach.

We believe the results can be improved by proposing a different scheme to select locks, for example using a more sophisticated memory abstraction. We also believe the analysis framework introduced in this paper is a good starting point to explore more sophisticated schemes and to deduce good optimizations that minimize the set of locks used to protect atomic sections (such as in [5]).

7. Related Work

Multi-granularity locking inference for shared databases The problem of multi-granularity locking and its related tradeoff between concurrency and overhead was first considered in the context of database management system [11]. The choice of locking granularity considered in the context of databases was based on the hierarchical structure of the data storage, e.g., fields, records, files, indices, areas and the entire database. The choice of locking granularity in our case is more challenging because of lack of any natural hierarchical scheme over the (possibly unbounded number of) memory locations accessed by a program. This requires creation of more sophisticated locking abstractions.

Hierarchical locking (simultaneous locking at various granularities) requires sophisticated locking protocols (as opposed to simply locking entities according to some total order on locks). Such protocols have been discussed in the context of data-base management systems [10]. In our work, we adapt these protocols for deadlock avoidance.

Lock inference for atomic sections There has been some recent work on compiler based lock inference from atomic section specifications. Some of these approaches either require programmer annotations or operate over a fixed (and finite) granularity of locks. On the contrary, our approach is completely automatic with support for multi-granularity locks.

The granularity of locks considered in [13] is one that is specified by programmer annotations. Our approach is completely automatic requiring no annotations from the programmer.

The granularity of locks considered in [9] is based on the (finite number of) abstract locations in the points-to graph. The lock associated with each abstraction location locks all (possibly unbounded number of) memory locations that are represented by that abstract location. Our more general multi-granularity locking scheme can use this abstraction as an instance of our locks. However, we also allow more fine-grained locking abstractions like expression locks.

The granularity of locks considered in [5] is based on path expressions (as described by a variable followed by a finite sequence of field accesses). The lock associated with each path expression locks all locations that the expression can ever evaluate to in any run and at any point of the program. Such a scheme is too coarse-grained compared to our seemingly similar, but quite different, expression locks. Our expression locks at a given program point p and in a given program run r, lock only the memory location to which the corresponding expression evaluates to at the program point p in the run r. Moreover, our expression locks are just an instance of our general multi-granularity locking scheme. However, the issue addressed in [5] is more about optimizing the set of locks that need to be acquired (since the cost of acquiring a lock is non-trivial) by phrasing it as an optimization problem. For example, if whenever x is accessed, y is also accessed, then we only need to acquire lock on y. This is an orthogonal issue and our work can also use leverage such an optimization.

8. Conclusions

We have presented general framework that infer locks to protect atomic sections. This framework is attractive for three main reasons. First, it provides an automatic implementation of atomic sections based on locking primitives, avoiding the disadvantages of optimistic concurrency. Second, it guarantees that the transformed programs respect the atomic semantics. And third, it is parameterized. It can be instantiated with different abstract lock schemes that better fit a user needs. We presented an implementation of our framework for a fixed lock scheme, and reported our experience when analyzing several benchmarks.

References

[1] Phoenix compiler infrastructure. http://research.microsoft.com/phoenix/.

[2] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*. Jun 2007.

[3] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. (http://www.cs.cornell.edu/∼siggi/papers/msr-tr07.pdf). Technical Report MSR-TR-2007-111, MSR, August 2007.

[4] Nir Shavit Dave Dice, Ori Shalev. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, Stockholm, Sweeden, September 2006.

[5] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL*, pages 291–296, 2007.

[6] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.

[7] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA*, page 102, 2004.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[9] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *TRANSACT*. June 2006.

[10] R. Lorie J. Gray and G.F. Putzolu. Granularity of locks in a shared database. In *VLDB*, pages 231–248, 1975.

[11] R. Lorie J. Gray, G.F. Putzolu, and I.L. Traiger. Granularity of locks and degrees of consistency. In *Modeling in Data Base Management Systems*, pages 364–394, 1976.

[12] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. In *TRANSACT*. Jun 2006.

[13] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, pages 346–358, 2006.

[14] T. Reps, S. Horowitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*. ACM, January 1995.

[15] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[16] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[17] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, St. Petersburg Beach, FL, Jan 1996.

[18] Joseph Uniejewski. SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Vol 1, Issue 1, SPEC, Fall 1989.

A. Program Semantics

We describe a program semantics using small step operational semantics relations. We use $\sigma = (H, \overline{L}, \overline{P}, \overline{in})$ to represent a state consisting of a shared store $H : \mathsf{Loc} \to \mathsf{Loc}$, a set of held locks $L_i \subseteq \mathsf{LNames}$ per thread, a command P_i per thread, and a boolean in_i telling whether the thread i is inside an atomic section. We abuse notation and also use σ to represent thread-local states $(H, \overline{L}, st, in)$. Essentially the program and in state are local, the heap and set of locks are global. We use $\mathsf{Loc} \supseteq \{\&x \mid x \in V\}$ to denote the domain of all program locations, which includes all stack locations too. The set LNames is the domain of lock names. We use an operator $+ : \mathsf{Loc} \times \mathbb{N} \to \mathsf{Loc}$ to compute the offset of a location, we assume that arbitrary pointer arithmetic is not supported, we omit these details for simplicity.

Figure 8 shows the semantic rules. The semantics non deterministically chooses a thread to perform a single step in the system. The language contains the constructs acquireAll and releaseAll that change the state of a thread to be inside and outside an atomic section. The acquireAll(N) command also acquires the locks in N, ensuring that N can be acquired concurrently with the locks held by the other threads. When reading expression values and performing assignments, the semantics first check whether the thread is inside an atomic section. If the thread is inside an atomic section, the semantics also check that all accesses are protected by a lock in L_i (see last rules in assignments and side-effect free expressions).

Cells allocated within an atomic section are automatically protected by the thread that creates them (using the runtime lock l_i^σ).

B. Soundness Proof

Here we include the proofs the soundness theorem presented in Section 4.2. All proofs assume we are given a sound abstract lock scheme Σ and a sound pointer analysis query *mayAlias*.

Definitions Our proofs use two functions *subs* and *reaches*, which we define in turn. The function $subs(e)$ returns, for a given expression e, all dereference subexpressions of e:

$$subs(e) = \begin{cases} \varnothing & e = \&x \\ \{*\&x\} & e = x \\ \{*e'\} \cup subs(e') & e = *e' \\ subs(e') & e = e' + i \end{cases}$$

The predicate $reaches(\sigma, v)$ answers if there exists a path in the state σ that reaches the location v from some program variable:

$$reaches(\sigma, v) = \begin{cases} \mathsf{true} & \langle \sigma, \&x \rangle \to v \\ reaches(\sigma, v') & H(v') = v \end{cases}$$

We write $\sigma @ p$ to say that σ is a state that reaches the program point p. If p is the point before a statement st, then $\sigma = (H, \overline{L}, \overline{P}, in)$ and $\exists st' \,.\, P_i = st; st'$.

Given a thread local state σ, we write $\langle \sigma, st \rangle \to \sigma'$ to denote a transition between states s.t. σ is at the program point before st ($\sigma @ b$) and σ' is at the program point right after st ($\sigma' @ a$). This transition may involve several small steps in our semantics.

Lemmas We proceed to prove the lemmas used in our theorem.

LEMMA 1. *The locks before any assignment protect the locations accessed directly by the statement. Formally,*

$$\begin{array}{l} \forall \sigma, st \,.\, \sigma @ (\bullet st) \Rightarrow \exists l \in transfer(*e_1 = e_2, N) \,. \\ \quad \langle \sigma, e_1 \rangle \to v \Rightarrow (\{v\}, \mathsf{rw}) \sqsubseteq [\![l]\!] \wedge \\ \quad \forall * e \in subs(e_i) \,.\, \langle \sigma, e \rangle \to v \Rightarrow (\{v\}, \mathsf{ro}) \sqsubseteq [\![l]\!] \end{array}$$

where $\sigma @ (\bullet st)$ is any state that reaches the program point before st, $subs(e_i)$ returns all dereference subexpressions of e_1 and e_2, e_1 ranges over $\{\&x, *\&x\}$.

Proof. The proof follows from the definition of *transfer* and the sets G included on each assignment. We show this holds for each assignment statement in our language: (a) $x = y$, (b) $x = *y$, (c) $*x = y$, (d) all other assignments.

(a) The assignment $x = y$ can be written as $*\&x = y$. To check the first condition, we consider $e_1 = \&x$. The transfer function includes $\overline{x}^{\mathsf{rw}}$ from the generating set G_x^{rw}. Since our abstract lock scheme Σ is sound, $\overline{x}^{\mathsf{rw}}$ protects the value of $\&x$ for writes.
To check the second condition we evaluate $subs(e_i) = \{*\&y\}$, so we need to check that the value $\&y$ is protected for reads. This is the case since $\overline{y}^{\mathsf{ro}} \in G_y^{\mathsf{ro}}$.

(b) This case, $x = *y$, is similar than the above. The first condition is proved in the same way. The second condition includes a new element since $subs(e_i) = \{*\&y, * * \&y\}$. The lemma holds because both $\overline{y}^{\mathsf{ro}}$ and $*\overline{y}^{\mathsf{ro}}$ are included in G_{*y}^{ro}

(c) On assignments $*x = y$ the first condition is slightly different, we check that $*\&x$ is protected for writes, which is the case because $*^{\mathsf{rw}}\overline{x}^{\mathsf{ro}} \in G_{*x}^{\mathsf{rw}}$. Additionally, for the second condition we have $subs(e_i) = \{*\&x, *\&y\}$, so we need to check that $\&x$ and $\&y$ are protected for reads. The former is protected by $\overline{x}^{\mathsf{ro}} \in G_{*x}^{\mathsf{rw}}$, the latter is protected by $\overline{y}^{\mathsf{ro}} \in G_y^{\mathsf{ro}}$.

(d) All other assignments, i.e. $x = \&y$, $x = y + i$, $x = \mathsf{new}$, $x = \mathsf{null}$, include a subset of the memory accesses seen in $x = y$. Thus, the proof is similar.

This concludes the proof of this lemma.

LEMMA 2. *Assume N is a set of locks protecting all locations accessed after a statement st. Except for unreachable locations, the locks inferred by $transfer(st, N)$ protect all the locations that were protected by N after the statement:*

$$\begin{array}{l} \forall st, l \in N, \sigma, v \in \mathsf{Loc} \,. \\ \quad \sigma @ (\bullet st) \wedge reach(\sigma, v) \wedge (\{v\}, \varepsilon) \sqsubseteq [\![l]\!] \Rightarrow \\ \quad\quad \exists l' \in transfer(st, N) \,.\, (\{v\}, \varepsilon) \sqsubseteq [\![l']\!] \end{array}$$

where $reach(\sigma, v)$ holds if the location v is reachable from some program variables in state σ.

Proof. Consider an assignment statement $e_1 = e_2$. Let l be a lock in N, let e be an expression such that $l = \widehat{e}_a$, where a is the program point after the statement. We need to consider two cases: (a) when e is not changed by the assignment, and (b) when the value of e changed.

(a) When e is unchanged we consider two more cases: some subexpression of e aliases e_1 or not. The first case yields that the assignment was a trivial assignment preserving the values in the store, we can consider this case as an instance of case (b) below. Hence, consider the other case, this is, e_1 is not aliased with any subexpression of e. Then $(\widehat{e}^a, \widehat{e}^b)$ appears in $closure(Id)$ and not in $closure(Q)$. Thus, $l' = \widehat{e}^b$ makes the lemma satisfiable.

(b) The value of e changes by the assignment, then consider two cases: (b1) $e_1 = x$, and (b2) $e_1 = *x$

(b1) In this case e must start with the variable x, otherwise the assignment wouldn't change e. Then we must consider three cases: (b1a) e_2 is one of the following y, $\&y$, $y + i$ or $*y$. (b1b) $e_2 = \mathsf{new}$, (b1c) $e_2 = \mathsf{null}$

(b1a) In this case, $e[e_2/x]$ is an expression that before the assignment has the same value as e after the assignment. Moreover, $(\widehat{e}^a, \widehat{e[e_2/x]}^b)$ appears in $closure(Id)$ and not in $closure(Q)$, thus $l' = \widehat{e[e_2/x]}^b$ makes the lemma satisfiable.

System step (step a single thread)

$$\frac{(H, \overline{L}, P_i, \overline{in}(i)) \rightarrow (H', \overline{L}, P_i', b') \quad \overline{P}' = \overline{P}[i \mapsto P_i'] \quad \overline{in}' = \overline{in}[i \mapsto b']}{(H, \overline{L}, \overline{P}, \overline{in}) \rightarrow (H', \overline{L}', \overline{P}', \overline{in}')}$$

Transformed atomic sections $\boxed{\mathsf{atomic}}$

$$\frac{\forall j, l \in N, l' \in L_j \,.\, \neg \mathit{conflict}(l, l')}{(H, \overline{L}, \underline{\mathsf{acquireAll}}(N), \mathsf{false}) \rightarrow (H, \overline{L}[i \mapsto N], \mathsf{skip}, \mathsf{true})}$$

$$\frac{}{(H, \overline{L}, \underline{\mathsf{releaseAll}}, \mathsf{true}) \rightarrow (H, \overline{L}[i \mapsto \varnothing], \mathsf{skip}, \mathsf{false})}$$

Control statements $\boxed{st; st, \mathsf{if}, \mathsf{while}}$

$$\frac{(H, \overline{L}, st_1, in) \rightarrow (H', \overline{L}', st_1', in')}{(H, \overline{L}, st_1; st_2, in) \rightarrow (H', \overline{L}', st_1'; st_2, in')}$$

$$\frac{}{(H, \overline{L}, \mathsf{skip}; st_2, in) \rightarrow (H, \overline{L}, st_2, in)}$$

$$\frac{\langle \sigma, b \rangle \rightarrow b', H', \overline{L}'}{\sigma = (H, \overline{L}, \mathsf{if}(b) st_1 \ \mathsf{else}\ st_2, in) \rightarrow (H', \overline{L}', \mathsf{if}(b') st_1 \ \mathsf{else}\ st_2, in)}$$

$$\frac{}{(H, \overline{L}, \mathsf{if}(\mathsf{true}) st_1 \ \mathsf{else}\ st_2, in) \rightarrow (H, \overline{L}, st_1, in)}$$

$$\frac{}{(H, \overline{L}, \mathsf{if}(\mathsf{false}) st_1 \ \mathsf{else}\ st_2, in) \rightarrow (H, \overline{L}, st_2, in)}$$

$$\frac{}{(H, \overline{L}, \mathsf{while}(b) st, in) \rightarrow (H, \overline{L}, \mathsf{if}(b) st; \mathsf{while}(b) st \ \mathsf{else}\ \mathsf{skip}, in)}$$

Assignments $\boxed{e_1 = e_2}$

Small step evaluation rules:

$$\frac{\langle \sigma, e_2 \rangle \rightarrow e_2', H', \overline{L}'}{\sigma = (H, \overline{L}, e_1 = e_2, in) \rightarrow (H', \overline{L}', e_1 = e_2', in)} \ e_2 \neq \mathsf{new}$$

$$\frac{\langle \sigma, e_1 \rangle \rightarrow e_1', H', \overline{L}'}{\sigma = (H, \overline{L}, e_1 = v, in) \rightarrow (H', \overline{L}', e_1' = v, in)} \ e_1 \neq *v'$$

Add locks for allocations within atomic sections:

$$\frac{H' = H \uplus \{v_1 \mapsto \mathsf{null}, ..., v_n \mapsto \mathsf{null}\} \quad L_i' = \begin{cases} L_i \cup \{l_i^\sigma\} \text{s.t. } (\{v_1, ..., v_n\}, \mathsf{rw}) \sqsubseteq [\![l_i^\sigma]\!] & in \\ L_i & \neg in \end{cases}}{\sigma = (H, \overline{L}, e_1 = \mathsf{new}(n), in) \rightarrow (H', \overline{L}', e_1 = v_1, in)}$$

Writes are protected within atomic sections.

$$\frac{in \Rightarrow (\exists l \in L_i \,.\, (\{v_1\}, \mathsf{rw}) \sqsubseteq [\![l]\!])}{\sigma = (H, \overline{L}, *v_1 = v_2, in) \rightarrow (H[v_1 \mapsto v_2], \overline{L}, \mathsf{skip}, in)}$$

Side-effect free expressions $\boxed{\langle \sigma, e \rangle \rightarrow e'}$

Small step evaluation:

$$\frac{\langle \sigma, e \rangle \rightarrow e'}{\langle \sigma, e + i \rangle \rightarrow e' + i} \ e \neq v \qquad \frac{\langle \sigma, e \rangle \rightarrow e'}{\langle \sigma, *e \rangle \rightarrow *e'} \ e \neq v$$

Values retrived, dereferences need read protection within atomic sections:

$$\frac{}{\langle \sigma, \&x \rangle \rightarrow \&x} \qquad \frac{v' = v +_\sigma i}{\langle \sigma, v + i \rangle \rightarrow v'}$$

$$\frac{in \Rightarrow (\exists l \in \overline{L} \,.\, (\{v\}, \mathsf{ro}) \sqsubseteq [\![l]\!])}{\langle \sigma = (H, \overline{L}, P, in), *v \rangle \rightarrow H(v)}$$

Figure 8. Small-step semantics of output language.

(b1b) In this case, e represents a heap path that doesn't exist before the assignment. This is because e is a sequence of dereferences, and its first dereference is a heap location allocated by this statement.

But we wonder if a location protected by $\widehat{e^a}$, is still protected by some l' before the assignment. We claim that this is true. Since e's value is reachable before the assignment, i.e. $\mathit{reach}(\sigma, v)$, it means that there is a non-empty set of expressions E, where each expression $e' \in E$ reaches v at the point before the assignment. All expressions in E don't go through the location allocated, otherwise they wouldn't reach v.

At some point later within the atomic section, an assignment must occur to make e reach the value v for the first time. Such assignment would have to be a store operation $*z = w$, where z is aliased with some subexpression e_s of e. At that point, the analysis must have included a lock that protects v using $e[w/e_s]$. Our lemma assumptions guarantee that w is protected by some lock $l = \widehat{w'}$ in N, then $e[w'/e_s] \in E$, and $\widehat{e[w'/e_s]}$ makes the lemma satisfiable.

(b1c) In this case, e is essentially an invalid expression since its source is x and x is null. But we have the same concern as with the previous case. The proof is identical.

(b2) In this case, the assignment is of the form $*x = y$. Some subexpression of e must alias x, otherwise the expression wouldn't change when updating $*x$. Let e' be the subexpression of e that aliases x. Then $e[y/e']$ before the assignment has the same value as e after the assignment. Since *mayAlias* is sound, $(\widehat{e^a}, \widehat{e[y/e']}^b)$ appears in $\mathit{closure}(S_{*x=y})$ and not in $\mathit{closure}(Q)$, thus $l' = \widehat{e[y/e']}^b$ makes the lemma satisfiable.

THEOREM 1. *Let σ be a reachable state, where thread i is about to execute a statement st within an atomic section. Let N be the result of our analysis for such atomic section. If at the entry of the atomic section, the thread i acquired all locks in N, then there exists some σ' such that $\langle \sigma, st \rangle \rightarrow \sigma'$, i.e. the program doesn't get stuck.*

Proof. The proof follows directly from the lemmas. The proof is a simple inductive argument on the structure of commands an atomic section. Our base case is when the atomic section is empty, which is trivially true with $N = \varnothing$. Each inductive case assumes that some sequence st already satisfies the theorem. The locks N' computed after adding new assigment st' before st still protect the accesses in st because of Lemma 2, but also the accesses in st' because of Lemma 1. Adding control structures, such as conditionals and while loops, uses the standard proof.