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.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.3.4 [*Processors*]: Compilers

General Terms Algorithms, Languages, Theory, Experimentation, Performance

Keywords Static lock inference, atomic sections, concurrency.

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 example, using locks to prevent data races. However, developing reliable and efficient applications with pessimistic concurrency is an arduous task. Programmers need to use fine-grain locks to minimize contention and need to carefully acquire and release locks to eliminate the possibility of a harmful data race and deadlocks. This is especially difficult to do in a modular way. Consider, for example, combining library code and client code, the 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.

Recently, programming languages' researchers have adopted the concept of *atomic sections* from the database community as a possible alternative. Atomic sections allow programmers to give a high level specification of the concurrency semantics. Without any intervention from the programmer, the underlying system enforces that atomic sections, i.e. sections of code protected by an *atomic* keyword, appear to be executed atomically. Under *strong atomicity* semantics these sections appear to be atomic with respect to any other statement in the program. Whereas under *weak atomicity* semantics atomic sections appear to be atomic with respect to other atomic sections in the code [2]. That is, for any program execution trace, there must exist an equivalent trace where all atomic sections are executed in some serial order. 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) [12, 11] or a software transactional memory (STM) that emulates such hardware [21, 9, 17]. 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 [20, 7] 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 some disadvantages. Certain applications do not perform well under an optimistic concurrency model as they incur a large number of transaction aborts and rollbacks. Additionally, not all atomic sections can be rolled back, for example after observable actions are performed. A well designed pessimistic approach could provide an alternative that avoids these disadvantages, by outperforming optimistic concurrency in applications that are not suited to transactions, and not performing too much worse than transactions in programs where transactions scale. In addition, a pessimistic approach allows observable actions inside atomic sections.

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 reads programs with atomic sections and produces equivalent programs that use lock for concurrency-control. The inferred locks enforce the weak atomicity semantics of the atomic sections. The transformed pro-

* This work was developed while the author was an intern at Microsoft Research, Redmond.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-59593-860-2/08/06…$5.00

```
 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:
14:
15:        x->next = y;
16:      }
17:    }
18: }
```

(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;
    acquire(&(x->next));
  }
  x->next = y;
}
releaseAll();
```

(b)

```
acquireAll({&(to->head),
           &(from->head), E});
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. `E` is a coarse lock protecting all elements in the `to` list.

grams can be run using a special lock runtime library. This approach allows users to write programs with atomic sections and run them 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 weak atomicity semantics originally 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 waste parallelism, 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 locations accessed within an atomic section. Second, to avoid deadlocks we borrow a locking protocol from the database community. Third, to reduce contention we insert locks of multiple granularities, using fine-grained locks as much as possible.

The following summarizes our contributions:

- We present formal definitions to model locks. We define the notion of *abstract lock schemes* that allows us 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 an input lock scheme specification.
- We show that our analysis is sound. That 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 describe a library to support locks of multiple granularities.
- We show experimental results for an instance of our framework indicating that the analysis scales well to medium size applications, yields better performance for benchmarks not suited to transactions, and performs not too much worse than an STM in some benchmarks where transactions scale.

The rest of this document is organized as follows. Section 2 illustrates the ideas behind our system using several examples. Section 3 introduces our formal definitions for locks and abstract lock 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 program that moves elements from one list to another. By the end of the function `move` the list `from` is empty and the list `to` contains the concatenation of the input lists. The entire function should be executed atomically, since it is wrapped in a block labeled with an atomic keyword. 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. This approach could introduce a lot of contention: the function `move` cannot be executed in one thread if any other thread is inside an atomic section.

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

Our system uses a third approach, which consists 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

```
 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.

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 an unbounded number of 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 the function `move`. 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 `E` 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 within an atomic section in terms of expressions that are in scope by the entry of the atomic block. We use the example in Figure 2 to illustrate this.

In Figure 2, the dereference of the variable `z` at line 10 could be protected by acquiring a lock on `&(*z)` (or, equivalently, just `z`) right before the access. Note that the variable `z` is not in scope at the entry of the section, because it is defined later at 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`. The expression `y->data` can be affected by the update of `x->data` at line 8 if `x` and `y` are aliased. In fact, `x` will alias `y` if the true branch at line 4 is taken. Since the compiler can't decide whether this branch is taken or not, it must consider both cases. When `x` and `y` are aliased, `x->data` will replace the value of `y->data`, and hence the location pointed by `z` at line 10 would be equivalent to `w` at the entry of the atomic section. Otherwise, when `x` and `y` are not aliased, `y->data` is not changed by the assignment in line 8, and thus `y->data` at line 7 would point to the location that `z` points to at line 10.

The compiler performs a similar backward tracing for every location accessed within the atomic section. Then, it introduces calls to acquire fine-grain locks for each expression derived at the entry of the section. For instance, in our example the compiler will lock both `y->data` and `w` ensuring that the access on line 10 is always protected by one of these locks. Additionally, to ensure termination the compiler bounds the size of the expressions it collects. The compiler introduces calls to acquire coarse-grain locks to protect expressions whose size exceeds the established bound.

3. Formalizing Locks

This section introduces formal definitions about locks. These definitions will be useful to answer questions involving locks and atomic sections. For example: What shared locations are protected by a lock? Are two locks protecting a common location? Are all shared accesses of an atomic section protected by a set of locks L?

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

Figure 3. Input Language

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 system. The formal definitions will allow us to show that our tool produces programs that respect the atomic semantics.

3.1 Language

Figure 3 presents our input language. The language contains standard constructs such as heap allocation, assignments, and control-flow constructs, but also includes atomic sections. Expressions include variables, dereferences, address-of variables, offsets, 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. We denote the domain of offsets by 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.

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.

3.2 Concrete Lock Semantics

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

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

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 $[\![l]\!] = (P, \varepsilon)$ 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 $[\![l]\!] = (\{v\}, \mathsf{ro})$ then l ensures that v is protected for reads, 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 protects a single memory location at all times, formally,

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

while a coarse-grain lock may protect more than one 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:

$$conflict(l_a, l_b) \Leftrightarrow \\ [\![l_a]\!] \sqcap [\![l_b]\!] \neq (\varnothing, __) \quad \wedge \quad [\![l_a]\!] \sqcup [\![l_b]\!] \neq (__, \mathsf{ro})$$

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

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

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 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:

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

Lock pairs We can also combine the power of two lock 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 locations 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

The lock semantics allows us to understand the relation between locks and locations. 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

$$\Sigma = (\mathcal{L}, \leq, \top, \bar{\cdot}_p^\varepsilon, +_p^\varepsilon, *_p^\varepsilon)$$

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 $\sqcup$, which 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:

$$\bar{\cdot}_p^\varepsilon : V \to \mathcal{L} \qquad +_p^\varepsilon : \mathcal{L} \times F \to \mathcal{L} \qquad *_p^\varepsilon : \mathcal{L} \to \mathcal{L}$$

where p is a program point in the domain PP and ε is an effect in Eff. The operator $\bar{\cdot}_p^\varepsilon$ takes a variable symbol and returns a lock name $l = \bar{x}_p^\varepsilon$; the operations $+_p^\varepsilon$ and $*_p^\varepsilon$ 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 inference algorithm. 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 $l_1 \leq l_2$, then the lock 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 $\bar{x}_p^\varepsilon$ protects the address of x to be used with the effect ε at the program point p, formally:

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

where $\sigma @ p$ denotes 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 to by the field i of v, then $l +_p^\varepsilon i$ must protect v', formally

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

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

- The operation $*_p^\varepsilon$ satisfies a similar condition:

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

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

The combination of $\bar{\cdot}_p^\varepsilon$, $+_p^\varepsilon$ and $*_p^\varepsilon$ 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 $\widehat{e}_p^{\,\varepsilon}$ be such lock, then:

$$\widehat{x}_p^{\,\varepsilon} = \bar{x}_p^\varepsilon \qquad \widehat{e+i}_p^{\,\varepsilon} = \widehat{e}_p^{\,\mathsf{ro}} +_p^\varepsilon i \qquad \widehat{*e}_p^{\,\varepsilon} = *_p^\varepsilon \ \widehat{e}_p^{\,\mathsf{ro}}$$

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:

$$
\begin{array}{lcl}
\mathcal{L} & = & \{l_e^p \mid \mathit{length}(e) \le k \wedge p \in \mathsf{PP}\} \cup \{\top\} \\
\le & = & \{(l_e^p, l_e^p), (l_e^p, \top) \mid l_e^p \in \mathcal{L}\} \\
\overline{x}_p^\varepsilon & = & \begin{cases} l_{\&x}^p & \text{if } k \ge 1 \\ \top & \text{if } k = 0 \end{cases} \\
l_e^{p'} +_p^\varepsilon i & = & \begin{cases} l_{e+i}^p & \mathit{length}(e+i) \le k \wedge p = p' \\ \top & \text{otherwise} \end{cases} \\
*_p^\varepsilon\, l_e^{p'} & = & \begin{cases} l_{*e}^p & \mathit{length}(*e) \le k \wedge p = p' \\ \top & \text{otherwise} \end{cases}
\end{array}
$$

The scheme consists of locks l_e for any expression of length equal or smaller than k. 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 [22]. Let A be the sets of 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:

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

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

$$
\begin{array}{lcl@{\qquad}lcl}
\mathcal{L} & = & \mathsf{Eff} & \overline{x}_p^\varepsilon & = & \varepsilon \\
\le & = & \sqsubseteq & l +_p^\varepsilon i & = & \varepsilon \\
\top & = & \mathsf{rw} & *_p^\varepsilon\, l & = & \varepsilon
\end{array}
$$

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

$$
\begin{array}{lcl@{\qquad}lcl}
\mathcal{L} & = & \{s \mid s \subseteq F\} & \overline{x}_p^\varepsilon & = & \top \\
\le & = & \subseteq & l +_p^\varepsilon i & = & \{i\} \\
\top & = & F & *_p^\varepsilon\, l & = & \top
\end{array}
$$

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:

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

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

In the earlier example from Figure 1(c) we used a locking scheme based on the Cartesian product of 3-limited expression locks and points-to set locks ($\Sigma_3 \times \Sigma_\equiv$). In the figure, we used a syntactic simplification to represent each lock. We used `to.head` and `from.head` to represent the fine-grain locks ($l_{\texttt{to.head}}, l_L$) and ($l_{\texttt{from.head}}, l_L$), where `L` is the points-to set for the list containers. We also used the symbol `E` to represent the coarse-grain lock $(\top, l_E)$, where `E` is the points-to set for all list elements. Notice that $(\top, l_E)$ is not a global lock, it can be held concurrently with any lock $(__, l_s)$ that protects any other points-to set $s \neq E$.

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 allow us to formally show that the analysis is sound. Then we discuss how we implemented an instance of this framework.

4.1 Analysis Framework

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 we saw in the example from Figure 2 when updating `x->data`. The result of the alias analysis is given by a relation $\mathit{mayAlias}(e_1, e_2, p)$ 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 an atomic section, we will compute a set of lock names $N_p \subseteq \mathcal{L}$. 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 the following 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 it reaches the entry of the atomic section.

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

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.

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:

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

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 location protected by $*\overline{x}$ after the statement is protected by $*\overline{y}$ before the statement. We express how other locks are transformed using the closure operator $\mathit{closure}(S)$. 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 $\mathit{closure}(Id)$ 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 $(*(*\overline{x}), *(*\overline{x})) \in \mathit{closure}(Q_x)$. Finally, all transfer functions include a new set of locks G that protect the locations accessed directly by the assignment.

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

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

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:

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

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

To illustrate how the framework 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 statement `z = *t2` defines z in terms of t_2, hence 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 alias analysis indicates that $mayAlias(t_1, y + data, \cdot)$ holds at that point, then the relation $*\overline{t_1} \sim *\overline{y} + data$ also holds. 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.

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

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

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

Function calls Figure 5 formulates how to reason about function calls. Essentially we model function calls as a composition of three groups of statements: the first group assigns actual arguments to the formal arguments used in the callee, the second group is the body of the callee, and the 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 a 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 $\underline{\mathsf{acquireAll}}(N)$ at the entry point of the atomic section, and a statement $\underline{\mathsf{releaseAll}}$ 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. States 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,*

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

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:*

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

where $reach(\sigma, v)$ *holds if the location* v *is reachable from some program variable 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 [6].

4.3 Implementation

We have developed an instance of this analysis framework for a fixed locking scheme and alias analysis. The formal presentation of our inference algorithm uses constructs, such as the closure operator and the $trans^*$ function, that can't be implemented in practice. This section describes how we implemented our framework and discusses optimizations that we performed for our chosen instance of the framework. We also discuss possible extensions to deal with pre-compiled libraries.

Implementing the framework Our implementation keeps a set of locks for each program point, and uses a 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 equations. The worklist contains pairs of locks and program points (l, p).

The worklist is initialized using the information in the sets G presented in Figure 4. That is, for each assignment $e_1 = e_2$ within the atomic section, we add (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.

The algorithm takes a pair (l, p) from the worklist, and for each predecessor statement st of the point p, we apply its transfer function and add the resulting locks $l'_1, ..., l'_n$ to the abstraction at the point p' before the statement. If the abstraction changed at p', we add (l'_i, p') to the worklist, and repeat the process until the worklist is empty.

We do not explicitly compute the closure operations used in the transfer functions and procedure calls. For transfer functions, our implementation is based on recursive substitutions of expressions. For function calls, we use a standard technique based on *function summaries* [19]. A function summary f_s essentially caches 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.

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

- We *map* the lock to the callee's context by analyzing 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.
- If no summary exists for l_1 in f_s, we add (l_1, exit_f) to the worklist, where exit_f is the program point by the end of f.
- If a summary for l_1 is found, then let l_2 be a lock in the result of the summary ($l_2 \in f_s(l_1)$).
- We *unmap* the lock l_2 back to the caller by modeling how actual arguments are passed as formals to the callee, i.e. $p_i = a_i$. The resulting locks are merged at the point p before the call, and added to the worklist if the abstraction changed.

Additionally, when the algorithm reaches the entry point of the function (l, entry_f) the analysis updates the function summary by adding l to $f_s(src(l))$, where $src(l)$ records the origin of a lock by the end of the function. We initialize $src(l) = l$ at the exit point of a function and we preserve it in the transfer functions, i.e, when $l' \in transfer(st, l)$ then $src(l) = src(l')$. After updating the summary, the analysis also *unmaps* the lock l to all callers of the current function.

Instantiating the framework We chose an abstract lock scheme based on k-limited expressions, points-to sets and read/write effects ($\Sigma_k \times \Sigma_\equiv \times \Sigma_\varepsilon$). We use Steensgard's analysis [22] to compute both the points-to sets in $\Sigma_\equiv$ and the *mayAlias* relation.

Our compiler implementation is specialized to take advantage of this scheme. In particular, from all possible pairs of expressions and points-to set locks, only few combinations need to be manipulated by the analysis. 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 has 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 expression locks (e, P) as children. These children protect a location within the memory partition protected by $(\top, P)$.

In practice, for a lock (e, P) the transfer functions will always conclude that P remains unchanged, because P is a flow-insensitive lock, and thus it 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 update this 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.

Supporting pre-compiled libraries Our compiler implementation assumes that the whole source code is available for analysis. However, pre-compiled libraries could be supported using specifications that summarize function effects. For instance, for our locking scheme based on $\Sigma_k \times \Sigma_\equiv \times \Sigma_\varepsilon$, we can use a list of coarse-grain locks as function specifications. Since our coarse-grain locks are flow-insensitive locks, they can essentially be used to protect all accesses done inside a pre-compiled function. When reasoning about calls to pre-compiled functions, the analysis needs to inspect the fine-grain locks inferred at the point after the call, evaluate if the expressions and subexpressions used in fine-grain locks could be changed by the call, and if so, replace the affected fine-grain locks by coarser locks. By specifying coarse-grain locks and effects in function specifications, our compiler would be able to do this.

5. Runtime System

The lock inference algorithm introduces locks of multiple granularities. This section presents a runtime library necessary to support this kind of locks.

5.1 Multi-Granularity Locking Library

Unlike traditional locks, with multi-grain locks there are pairs of locks that cannot be held concurrently. Hence, acquiring multi-grain locks in any linear order 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. [16, 15] 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 $l_b \leq \top$.

(a)

	X	S
X		
S		✓

(b)

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

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

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, the current 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) dependencies 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 child 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), each ancestor 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 each ancestor l' ($l \leq l'$) must be held by this thread in SI_x or I_x mode.
- Siblings are acquired in the same order by all threads.
- 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 for our locking scheme ($\Sigma_k \times \Sigma_\equiv \times \Sigma_\varepsilon$) in a runtime library. The library's API consists of three functions: *to-acquire*, *acquire-all*, *release-all*. The function *to-acquire* takes a *lock descriptor* (see below) and adds it to a list of pending locks. The function *acquire-all* proceeds to request all pending locks using the protocol presented above. The function *release-all* unlocks every lock in the list and clears the list.

In order to acquire locks using the protocol, our library requires partial 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 do not store the entire lock structure in the library, we provide the library with the relevant portion of the locking structure using *lock descriptors* instead. For our locking scheme, the lock descriptor is just a triple consisting of a memory address (describing the Σ_k component), a number (describing the $\Sigma_\equiv$ component), and a boolean (indicating the effect ro or rw). Internally, the library maintains a map associating lock descriptors with actual locks.

The transformation we presented in Section 4 inserts statements acquireAll(N) for a set of locks $N = \{l_1, ..., l_n\}$, and releaseAll to release all locks. Our implementation translates releaseAll into *release-all*, and acquireAll(N) 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.

5.3 Supporting nested atomic sections

Our inference algorithm and transformation do not need to reason about nesting of atomic sections. When atomic sections are nested, the outer section protects the locations included in the inner section, hence it is unnecessary to acquire locks when entering the inner section. However, sections could be nested in one thread, but not nested in another thread, more precisely, an inner section in one thread can be the outer-most section of some other thread. Such other thread must acquire locks when entering that section.

Nested sections can be supported via an additional form of context sensitivity in the program: whether the current thread is inside an atomic section. This can be implemented with a small extension to our runtime library by adding an integer variable `nlevel` for each thread. We can track the nesting level of each thread by incrementing (decrementing) `nlevel` whenever acquireAll (releaseAll) is called. However, we would only append, acquire, and release locks when the value of `nlevel` is 0.

6. Results

This section describes our experimental setup and presents both compile-time and run-time statistics.

6.1 Experiment Setup

We used the Phoenix infrastructure [1] as a front end to implement our analysis. 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 performs the program transformations based on the analysis results. All phases are implemented in C#. We manually replicate the transformation phase in order to generate the transformed programs using a different compiler (see further below).

We used several values for k, between 0 and 9, to build the k-limited expressions. Programs were compiled on a 1.66Ghz Dual Core, 2 GB RAM machine, running Windows XP SP2. Runtime experiments were performed on a 1.86Ghz Intel Xeon dual-quad core, 16 GB RAM, server system running Windows Server 2003 SP2 x64 edition.

Benchmarks We analyzed several of the SPECint2000 benchmarks [23], the STAMP benchmarks (v0.9.6) [4] and a set of micro-benchmarks using traditional data-structures. These sets of programs correspond to the top, middle and bottom portion of Table 1, respectively. The STAMP and micro-benchmarks are concurrent applications that contain atomic sections protecting shared memory accesses. The first three micro-benchmarks are implementations distributed with the STAMP benchmarks. The *hashtable-2* is a different implementation of hashtable. The difference between *hashtable* and *hashtable-2* is how put operations are implemented. Given a list in a bucket, *hashtable-2* inserts a new entry at the beginning of the list, *hashtable* traverses the list to perform the insertion. Additionally, *hashtable-2* never changes the size of the table or rehashes values. On the other hand a put in *hashtable* might rehash and access all elements in the table during this process. These four micro-benchmarks are run with the same harness that performs several operations (*put* or *insert*, *get* or *lookup*, and *remove*). Each op-

Program	Size (Kloc)	Atomic sections	Analysis Time (s) $k=0$	$k=9$
gzip	10.3	1	1.6	3.4
parser	14.2	1	4.2	11.0
vpr	20.4	1	5.0	32.6
crafty	21.2	1	5.3	111.3
twolf	23.1	1	6.2	15.4
gap	71.4	1	3.0	76.6
vortex	71.5	1	10.6	193.7
vacation	10.1	3	2.4	3.3
genome	9.8	5	1.7	2.0
kmeans	4.2	3	1.4	1.4
bayes	11.6	7	1.8	2.4
labyrinth	8.0	3	1.2	1.3
hashtable	3.4	4	0.7	0.8
rbtree	1.9	4	0.8	0.8
list	1.5	4	0.7	0.8
hashtable-2	0.4	4	0.7	0.7
TH	5.1	7	1.0	1.0

Table 1. Program size and analysis time in seconds.

eration 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 program *TH* accesses two data-structures instead of one. It essentially combines *rbtree* and *hashtable*. Like all other micro-benchmarks, this program is run with a harness that performs several *put*, *get* and *remove* operations. However, each operation randomly selects to use one or the other data-structure, hence half of the accesses are on each structure.

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 (v0.9.3) STM [7], distributed with the STAMP benchmarks, to compare the runtime performance of our approach against an optimistic alternative. The TL2 STM can be compiled under Windows using Cygwin and gcc-3.4.4.

Manual transformations To make comparisons fair, we use the same compiler (*gcc*) to build both the programs with TL2 and the transformed programs with our multi-grain locking library. Because the Phoenix infrastructure doesn't support source-to-source transformations, we manually performed our transformations at the source level in order to compile the programs with *gcc*. Notice, our manual intervention is minimal and mimics the same changes done by our compiler in Phoenix. With the appropriate infrastructure, these changes could be done completely automatically.

6.2 Compiler Statistics

Table 1 shows the size of each program and the analysis time. The analysis time corresponds to the second phase of our compilation process. This includes the time for the unification-based points-to analysis and the backward dataflow analysis, but doesn't include time spent parsing or generating code. 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 this column can be used as a rough estimate of the time spent in the pointer analysis.

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. The STAMP benchmarks and the micro-benchmarks 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. On average, the analysis is an order of magnitude faster than parsing the programs using Phoenix.

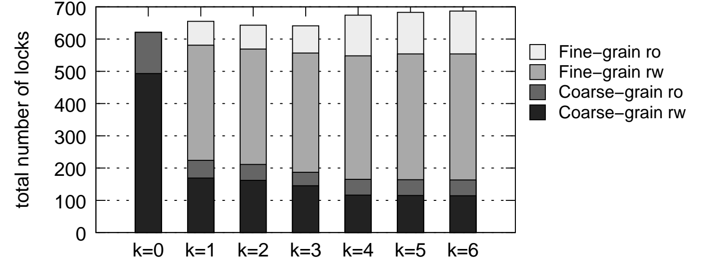

Figure 7. Combined total number of fine-grain and coarse-grain locks from all programs. Increasing the analysis precision reduces the number of coarse locks and possibly reduces contention.

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 into four categories: (a) fine-grain read-only locks (read-only expressions), (b) fine-grain read-write locks, (c) coarse-grain read-only locks (read-only points-to sets), (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.

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-grain 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 atomic sections: these objects are not reachable by the entry of their atomic section, and thus, they are not shared unless they become explicitly stored in another location. Locks protecting the other location implicitly protect the allocated cells. The dataflow analysis deduces this when tracing fine-grain locks up to their allocation site.

Beyond $k = 6$ 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 global variables, structure fields, or array entries. Non-recursive structures have a bounded depth, and typically programmers use a depth of 2 or 3 heap dereferences. Both offset operations and heap dereferences contribute to the length of an expression, thus many expressions with 3 heap dereferences may have length $k = 6$.

6.3 Runtime Statistics

We evaluate the runtime performance only on the concurrent applications (STAMP and micro-benchmarks). We ran the STAMP benchmarks using the low contention parameters suggested in the documentation distributed with the programs. For the micro-benchmarks we used two parameter configurations: *low* and *high*. The *low* setting reduces contention by performing more read-only operations, in particular, *gets* are four times more common. Conversely, the *high* setting uses *puts* four times more often. Except for *hashtable-2*, the *high* setting introduces more contention than the *low* setting. In *hashtable-2*, a *put* operation updates a single shared memory location, hence the *high* setting does not increase the contention as much as in the other applications.

Program	Global	Coarse ($k = 0$)	Fine + Coarse ($k = 9$)	STM
	Execution time (s)			
genome	8.6	9.0	14.5	15.2
vacation	0.8	0.8	0.8	263.3
kmeans	49.3	52.7	76.9	111.2
bayes	49.8	49.9	49.6	82.9
labyrinth	7.8	7.9	7.8	4.1
hashtable-high	51.7	51.3	51.4	75.8
hashtable-low	44.8	23.1	23.1	11.1
rbtree-high	41.1	40.3	40.3	5.3
rbtree-low	40.2	20.9	20.9	5.0
list-high	49.5	48.6	48.6	19.2
list-low	43.4	22.3	22.3	8.8
hashtable-2-high	41.2	40.3	20.6	5.1
hashtable-2-low	40.1	21.1	20.6	5.0
TH-high	45.4	24.5	24.5	53.0
TH-low	41.3	11.4	11.3	7.3

Table 2. Execution times using 8 threads.

Table 2 shows the running time of the evaluated benchmarks using 8 threads. The first column shows the running time using a single global lock to protect each atomic section. The next two columns show the time consumed by the transformed applications when using the locks chosen by the analysis with $k = 0$ (Only Coarse) and with $k = 9$ (Coarse + Fine). The last column shows the running time when using TL2. Figure 8 shows some scalability tests for selected applications.

Runtime impact of multi-granularity locks. The benefits of our transformation considerably vary between applications. In particular, our transformation has a negative effect on the STAMP benchmarks. These programs have no opportunity of increasing parallelism by distinguishing read and write effects or by introducing our multi-grain locks. Compared to global locks, our transformation will increase the execution times due to the overhead in the multi-grain locking protocol. This negative effect is illustrated in *genome*, shown in Figure 8. In this program, the *Coarse+Fine* configuration protects 4 atomic sections using a single coarse-grain lock with write effects, which enables the same parallelism than a global-lock. Only one section is protected by fine-grain locks. We do not see these locks improving parallelism either, but instead we see an increase in overhead: the program acquires more locks and the multi-grain locking protocol performs more operations.

The opposite effect was observed in the micro-benchmarks. In *rbtree*, for example, we can see the benefits of tracking read and write effects. The analysis determined that *get* operations perform only memory reads. This allows our system to run multiple *get* operations concurrently by protecting them using read-only locks. Since the *low* configuration performs more *get* operations, the transformation with coarse-grain locks runs almost twice as fast than using global locks. For the same reason, coarse-grain locks are twice as fast in the *low* setting than in the *high* setting. The analysis with $k = 9$ didn't introduce any fine-grain locks, hence the results are the same as with $k = 0$.

The benefit of fine-grain locks over coarse-grain locks is revealed in *hashtable-2* (Figure 8). As mentioned earlier, the *put* operation in this program only updates a single shared memory location (one bucket entry). The analysis with $k = 9$ assigns a single fine-grain lock to protect that location. When *put* operations are four times more common than other operations (*high* setting), fine-grain locks halve the execution time of coarse-grain locks.

Our technique enabled parallelism between pairs of *put*s (using fine-grain locks) and between pairs of *get*s (using read-only locks), but *put* and *get* still have contention with each other when using locks. We believe this is the reason why the performance of multi-grain locks did not improve in *hashtable-2* when moving from 4 to 8 threads. Since four out of five operations are *puts*, it is highly likely (with more than an 80% chance) that one of the 8 threads is performing a *put* while another thread is performing a *get* or a *remove*, and hence some threads are likely to become blocked and waste parallelism.

Our system's performance improves when transactions access disjoint portions in memory that are protected by independent locks. Two accesses to disjoint data-structures can always run concurrently. This effect is illustrated in *TH*, also shown in Figure 8. Since this program uses two data-structures, coarse-grain locks can always exploit more parallelism than a global lock. When using 8 threads, our inferred locks are 1.9 times better than a global lock in the *high* setting, and 3.6 times better in the *low* setting.

Comparison with TL2 Except for *labyrinth*, the TL2 system performs worse in the STAMP benchmarks than using either a global lock or our inferred multi-grain locks. This is because a lot of time is spent in rolling back and re-executing atomic sections with conflicts. This is clearly illustrated in *vacation*, which runs only 1,000 successful transactions, while it aborts a total of 1.7 million transactions. In *genome*, the overhead of acquiring fine-grain locks makes our system only 5% faster TL2, in contrast, coarse-grain locks are 41% faster. In *kmeans*, our programs also run about 29% faster than TL2, even with the overhead introduced by fine-grain locks.

The TL2 system performs better on the micro-benchmarks when using the *low* setting. It also performs well on for *rbtree*, *hashtable-2* and *list* with the *high* setting. On average, when considering all micro-benchmarks together, TL2 ran 14% faster than the multi-grain locks generated with $k = 9$ in the *high* setting, and 60% faster in the *low* setting. This is expected, as locks cannot model as much parallelism as an optimistic system. TL2 scales better than our inferred locks in *hashtable-2*, even when our compiler uses fine-grain locks. We believe that the additional speed in TL2 comes from parallelism between pairs of operations that are disallowed by our locks, such as pairs of *put* and *get* operations.

In *hashtable*, a *put* operation might resize the table and re-hash values, in which case the operation will access the entire table. For this reason TL2 spends lots of time rolling back transactions in *hashtable-high*. We see a similar effect in *TH-high*, since one of the data-structures in *TH* is *hashtable*. Interestingly, when reaching 8 threads in *TH-high*, our multi-grain locks scales well, but TL2 becomes slower than using a global lock.

Final comments We have seen that our compiler can scale to analyze large atomic sections. While the results vary across benchmarks, we have also seen that our transformation can effectively exploit more parallelism than a global lock. We believe our results can be improved by proposing new locking schemes to select locks. For example, in *kmeans* an array-range analysis could detect when atomic sections only access a portion of a matrix. The analysis framework introduced in this paper provides 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 (like in [8]).

Overall, our system is preferable to global locks when applications have low contention (e.g. in *rbtree-low*) or have several sections accessing disjoint data-structures (e.g. in *TH*). Optimistic approaches are likely to be more efficient when contention is low. Our system would be preferable to STMs in three scenarios: when applications have non-reversible operations within atomic sections, have high contention, or have long atomic sections. In the last two scenarios, an optimistic approach might introduce large overhead in detecting conflicts and rolling back transactions (e.g. *vacation*,

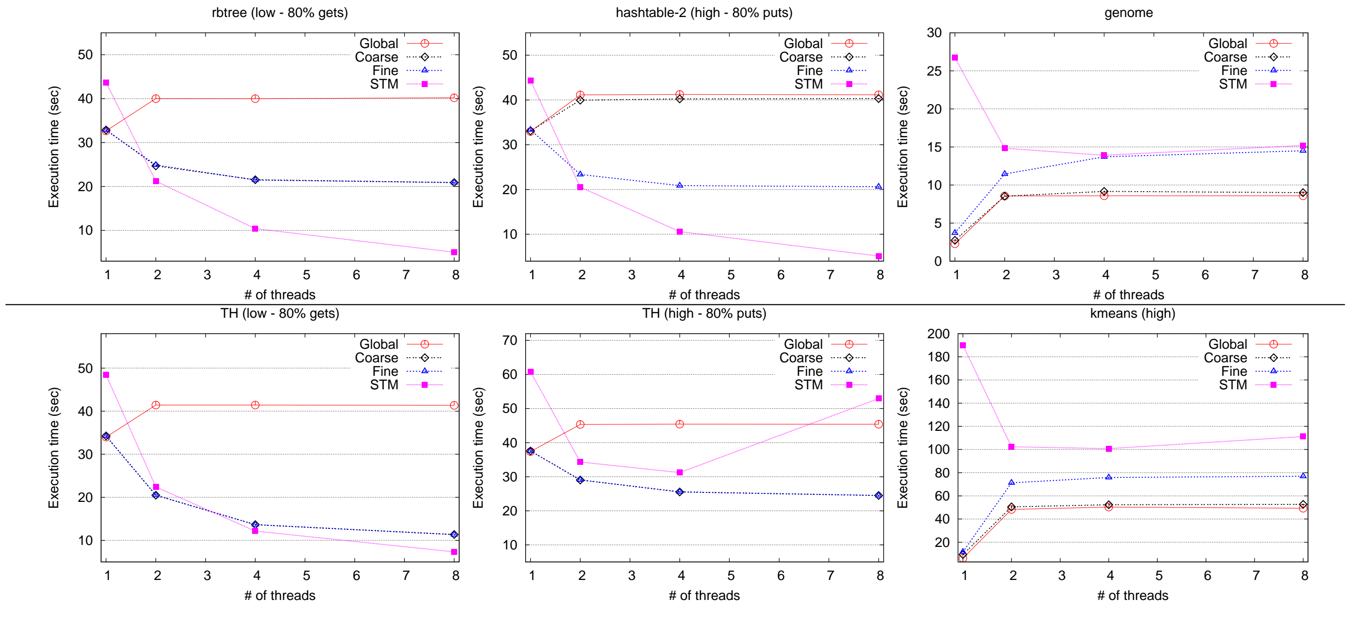

Figure 8. Execution times for *rbtree*, *hashtable-2*, *TH*, *genome*, and *kmeans* using 1, 2, 4 and 8 threads.

hashtable-high). Our system is preferable to both global-locks and STM in applications with low contention and non-reversible operations, but also in applications where several atomic sections access disjoint data-structures with very high contention (e.g. *TH-high*).

7. Related Work

Multi-granularity locking The problem of multi-granularity locking and its related trade-off between concurrency and overhead was first considered in the context of database systems [16]. 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 complex locking abstractions. Multi-grain locking requires sophisticated locking protocols, as opposed to simply locking entities according to some total order. Such protocols have been discussed in the context of database systems [15]. 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 specifications. Some of these approaches either require user annotations or operate over a fixed (and finite) granularity of locks. On the contrary, our approach is automatic and supports multi-granularity locks.

The granularity of locks considered in Autolocker [18] is one that is specified by programmer annotations. Our approach is completely automatic requiring no annotations from the programmer other than atomic sections. The granularity considered by Hicks et al. [13] is based on the (finite number of) abstract locations in a points-to graph. The lock associated with each abstraction location locks all memory locations that are represented by that abstract location. Our more general locking scheme framework can be instantiated using this lock abstraction. However, we also allow more fine-grained locking abstractions like expression locks.

The granularity of locks considered by Emmi et al. [8] is based on path expressions (a variable followed by a finite sequence of fields). The lock associated with each path expression locks all locations that the expression can ever evaluate to in any run and at any program point. 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 by Emmi et al. [8] 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 leverage such an optimization.

The granularity of locks considered by Halpert et al. [10] is based on components of interfering atomic sections. They reduce contention by detecting when sections do not interfere. They can use a fine-grain lock to protect a component. However, all sections in a component must use the same lock, thus their support for fine-grain locking is restricted. For example, in *hashtable-2* this would require using the same lock to protect both *get* and *put*, hence not allowing the fine-grain locking of *put*.

Hindman and Grossman [14] present a source to source translation to implement atomic sections in Java using locks. They delay acquiring locks until the first time an access is encountered. This allows them to use fine-grain locking more freely, but unlike our approach, it doesn't prevent deadlocks ahead of time. Instead they log write operations and support rolling back transactions when deadlocks are detected. Our system prevents aborting transactions, and hence supports transactions with non-reversible operations.

Other approaches for concurrency specifications Besides using atomic sections, several researchers have looked at other models to specify constraints on concurrent applications.

Vaziri et al. [24] present a data centric synchronization approach. Programmers only label data that must be accessed together in order to maintain data consistency, then a static analysis infers critical sections to enforce these consistency requirements. Colorama [5] introduces a hardware alternative to infer these sections.

Both Vaziri et al. and Colorama use transactions to implement critical sections. This paper addresses a complementary issue: how to implement efficiently these critical sections using locks. Specifications like those required by Varizi et al. could help our compiler minimize the set of locks used to protect an atomic section, because only a few memory accesses must be protected to maintain consistency. This is an interesting direction for future work.

Flux [3] presents a different mechanism for writing concurrent applications based on high-level data flow between computation nodes (commonly C/C++ functions). Programmers declare mutual exclusion between these nodes by specifying lock names and their effects (read or write). The Flux programming model is more manageable for programmers than traditional lock APIs, in particular, the language provides a type system that can detect and prevent deadlocks. However, the programmer is still responsible for associating locks with data, choosing an appropriate granularity level for their locks, and ensuring that all shared accesses are protected. In contrast, all these tasks are done automatically by our compiler.

8. Conclusions

We have presented a general framework that infers 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 to fit user needs. We presented an implementation of our framework for a fixed lock scheme and reported our experimental experience.

Acknowledgments

We would like to thank Krishnaprasad Vikram for his helpful remarks about multi-grain locking protocols. We would also like to thank the anonymous reviewers for their useful comments.

References

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

[2] Colin Blundell, E. Lewis, and Milo Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[3] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux: a language for programming high-performance servers. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 13–13, 2006.

[4] 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 *Proceedings of the International Symposium on Computer Architecture*, Jun 2007.

[5] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 133–144, 2007.

[6] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. Technical Report MSR-TR-2007-111, MSR, August 2007.

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

[8] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 2007.

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

[10] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2007.

[11] 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 *Proceedings of the International Symposium on Computer Architecture*, 2004.

[12] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, 1993.

[13] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[14] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, October 2006.

[15] R. Lorie J. Gray and G.F. Putzolu. Granularity of locks in a shared database. In *Proceedings of International Conference on Very Large Databases*, 1975.

[16] 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, G.M. Nijssen ed., North Holland Pub.*, 1976.

[17] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.

[18] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 346–358, 2006.

[19] T. Reps, S. Horowitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. ACM, January 1995.

[20] B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, 2006.

[21] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the ACM symposium on Principles of Distributed Computing*, 1995.

[22] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, Jan 1996.

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

[24] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, January 2006.