Autolocker: Synchronization Inference for Atomic Sections *

Bill McCloskey   Feng Zhou

UC Berkeley

{billm,zf}@cs.berkeley.edu

David Gay

Intel Research

david.e.gay@intel.com

Eric Brewer

UC Berkeley and Intel Research

brewer@cs.berkeley.edu

Abstract

The movement to multi-core processors increases the need for simpler, more robust parallel programming models. Atomic sections have been widely recognized for their ease of use. They are simpler and safer to use than manual locking and they increase modularity. But existing proposals have several practical problems, including high overhead and poor interaction with I/O. We present pessimistic atomic sections, a fresh approach that retains many of the advantages of optimistic atomic sections as seen in "transactional memory" without sacrificing performance or compatibility. Pessimistic atomic sections employ the locking mechanisms familiar to programmers while relieving them of most burdens of lock-based programming, including deadlocks. Significantly, pessimistic atomic sections separate correctness from performance: they allow programmers to extract more parallelism via finer-grained locking without fear of introducing bugs. We believe this property is crucial for exploiting multi-core processor designs.

We describe a tool, Autolocker, that automatically converts pessimistic atomic sections into standard lock-based code. Autolocker relies extensively on program analysis to determine a correct locking policy free of deadlocks and race conditions. We evaluate the expressiveness of Autolocker by modifying a 50,000 line high-performance web server to use atomic sections while retaining the original locking policy. We analyze Autolocker's performance using microbenchmarks, where Autolocker outperforms software transactional memory by more than a factor of 3.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Languages, Algorithms

Keywords Atomic, Lock, Pessimistic

1. Introduction

Writing parallel and concurrent systems programs is a notoriously difficult task. We propose a new programming concept called a *pessimistic atomic section* to mitigate this problem. We call them pessimistic because they use locks rather than optimistic concurrency

```
struct entry { int k; int v; struct entry *next; };

mutex table_lock;
struct entry *table[SZ] protected_by(table_lock);

void put(int k, int v) {
  int hashcode = ...;
  struct entry *e = malloc(...);
  e->k = k; e->v = v;
  atomic {
    e->next = table[hashcode];
    table[hashcode] = e;
  }
}

int get(int k) { ... }
int delete(int k) { ... }

void f(int a, int b) {
  atomic { put(a, 0); put(b, delete(a)); }
}
```

Figure 1. A hash table implemented using atomic sections.

control. Semantically, pessimistic atomic sections are guaranteed to execute atomically, much like database transactions. They are translated into normal synchronization operations, so they integrate well with existing code. Unlike manual locking, they are composable, so they support a more modular style of programming. Finally, the performance of a pessimistic atomic section can be tuned without affecting the correctness of the underlying code. This paper evaluates pessimistic atomic sections via a prototype compiler called Autolocker that uses a provably sound algorithm to translate atomic sections written in C into traditional synchronization primitives. Autolocker handles many of the tedious and error-prone details of parallel programming while retaining programmer control over performance.

Autolocker is not a radical departure from traditional lock-based synchronization. Every piece of data that is shared by multiple threads must be protected by a lock. Autolocker requires programmers to write annotations that connect locks to shared data. Based on these annotations, Autolocker transforms code to acquire a lock when its associated data is accessed from within an atomic section. In accordance with the policy of strict two-phase locking, all locks are released at the end of atomic sections and not before (ch 17.1 of [29]). Thus, Autolocker guarantees that data remains consistent at all times. By ensuring that locks are acquired in a well-defined order with no cycles, it also guarantees the absence of deadlocks.[1]

1.1 Example

We begin the description of the Autolocker tool with an example that illustrates its use. Figure 1 shows a hash table implementation

---

[1] It is a well-known fact that an acyclic locking order is a sufficient condition to guarantee deadlock freedom.

written using atomic sections. Only the put operation is shown; the get and delete operations are written in a similar fashion.

The semantics of atomic sections mean that they appear to execute serially, without interruption, like database transactions. In other words, they run as if a single, global lock bracketed every atomic section in the program so that only one could execute at once. In fact, using a single lock would be a valid but inefficient implementation strategy. Instead, Autolocker permits programmers to specify a more efficient locking policy.

This hash table uses a single lock, `table_lock`, to protect every element. The `protected_by` annotation protects all heap data reachable from a variable (until a different annotation is reached). Thus, every entry in the hash table is protected by `table_lock`. Here is how Autolocker compiles the hash table's put function:

```
void put(int k, int v) {
  ... non-atomic operations ...

  begin_atomic();
  acquire_lock(&table_lock);
  e->next = table[hashcode];
  table[hashcode] = e;
  end_atomic();
}
```

Autolocker includes a library of runtime functions to implement `begin_atomic`, `end_atomic`, and `acquire_lock`. The first two functions can be nested—they do nothing for inner atomic sections. The `acquire_lock` function acquires a given lock if it isn't held already. Locks are released when the outermost `end_atomic` function is called.

It might appear that Autolocker provides few benefits for such a simple program, since a competent programmer would have little difficulty recognizing when `table_lock` should be acquired in the program above. However, Autolocker offers three key advantages: it simplifies modular programming, it gives programmers the ability to improve performance without affecting correctness, and it reduces the likelihood of programmer error.

Modularity. Atomic sections are composable in Autolocker. For example, let us pretend that the hash table operations reside in a library which is called by the client function `f` in Figure 1. Since `f` performs *two* hash table operations atomically, it cannot rely on the internal locking performed by the put and delete operation. Without Autolocker, the hash table library would have to expose functions to lock and unlock itself. Besides violating abstraction, these functions are very easy to misuse to create race conditions or deadlocks. Deadlocks are particularly likely, since the programmer is unable to order lock acquisitions consistently without understanding how locking works within the hash table library.

Autolocker solves these difficulties. It has knowledge of the whole program, including the hash table implementation, so it knows which locks must be acquired when the atomic section in `f` begins. It also determines a single lock acquisition order for the entire program, which prevents deadlocks. For modules without source code, annotations can be placed at the module boundary to ensure proper locking of data used by external code, although this feature is only partially implemented.

Control. The simple locking policy shown for this hash table performs well if there is little thread contention, but increased contention calls for a more fine-grained locking policy. Pessimistic atomic sections have an extremely important property: performance can be tuned without introducing deadlocks or data races. The most important performance trade-off of locks is between a coarse-grained locking policy and a fine-grained one. A coarse-grained policy uses a smaller number of locks to protect a larger part of the heap. This policy generates few lock acquisitions (less overhead) but may lead to heavy lock contention (more waiting for locks). Fine-grained policies use more locks to protect data, so there is a lower likelihood that two accesses will acquire the same lock (less waiting). However, more locks are acquired overall, which can be inefficient if acquiring a lock has significant overhead.

Using manual locking, changing granularity is very painful. Each modification induces wide-ranging changes across the code-base, since each acquisition of the affected locks must be updated. Even worse, it is very easy to introduce deadlocks when making a locking policy more fine-grained, since programmers often fail to reason about changes in the order of lock acquisition.

When using Autolocker, the locking policy is determined entirely by lock protection annotations. By introducing lock variables and changing data protections, the programmer can make the locking policy more or less fine-grained. These changes do not affect the functional part of code, which is written using atomic sections and makes no mention of locks. Since the functionality remains the same, Autolocker ensures that correctness is unaffected. We prove the soundness of Autolocker's algorithm later in the paper.

Using Autolocker, only two declarations in the hash table example must be changed to make its locking policy fine-grained:

```
struct bucket {
  mutex lock;
  struct entry *head protected_by(lock);
};

struct bucket table[SZ];
```

In this version of the code, two elements with different hash codes are protected by different locks. Despite this change, the code that implements the hash table operations is unaffected. In our experiments, this fine-grained hash table will perform significantly better than the previous one under heavy contention.

Safety. Unfortunately, the new version of the code is unsafe. Autolocker will complain that under this new locking policy the operations performed by `f` may deadlock. When `f` executes, it will first acquire the lock for whichever hash code the key `a` maps to. Later, it will acquire the lock for `b`. There is no *a priori* ordering of these two lock acquisitions with respect to each other. If two threads execute simultaneously, one calling `f(x, y)` and the other calling `f(y, x)`, they could end in a deadly embrace. The solution is to use *multi-granularity locking*, a concept borrowed from databases [14]. We describe multi-granularity locking in Section 5.4. Essentially, it permits Autolocker to acquire a single mutual exclusion lock that protects the entire hash table or to acquire a more fine-grained per-bucket lock when only one bucket is accessed. This arrangement is free of deadlocks.

Besides being a useful debugging technique, Autolocker's ability to detect potential deadlocks is crucial to its correctness. Because the tool assumes complete responsibility for acquiring locks, the programmer loses control over how and in what order the locks are acquired. If Autolocker did not guarantee deadlock freedom, the programmer would have no way of doing it herself. A bad decision made by the tool would lead to a deadlock that could not be corrected by the programmer. Thus the guarantee of deadlock freedom is crucial to the usability of Autolocker.

1.2 Transformation Process

Converting a program that uses atomic sections to one that uses locks is a multi-step process. We describe these steps in turn throughout the rest of the paper. In brief, they are:

1. Merge all files into a single program. Extract outermost atomic sections from the program, even across function boundaries.
2. Use the Autolocker type system to transform the code for each atomic section into a *computation history* (Section 3.1). A com-

putation history describes how an atomic section behaves with respect to synchronization. In particular, it contains all the locks that are needed by the code.

3. Process the computation histories to generate a graph of dependencies among the locks that are required by a history (Section 4.1). If the dependencies are cyclic, warn of a possible deadlock. Otherwise, use topological sort to generate a global, deadlock-free ordering in which to acquire locks.

4. Insert lock acquisitions where necessary, being sure to respect the ordering that has been computed (Section 4.2). Remove any redundant acquisitions.

There are two main contributions in this paper. The first is the algorithmic framework described above, including the type system to generate computation histories, the program transformation, and the proof of correctness (Sections 2-4). The second is the Autolocker tool itself, which operates on real C programs using extensions described in Section 5. In Section 6, we evaluate the success of Autolocker on several benchmarks, including a large one of over 50,000 lines to evaluate expressiveness and two smaller microbenchmarks to validate performance. We were generally pleased with expressiveness, and performance was excellent (a 3x improvement over software transactions in one case). We discuss related work in Section 7 and conclude in Section 8.

2. Analysis

In this section, we look at some of the challenges facing pessimistic atomic sections and then compare them with optimistic atomic sections. Some of these challenges are fundamental, such as the need for annotations, while others are limitations of the current implementation.

2.1 Caveats

Annotations. The most obvious caveat of pessimistic atomic sections is the requirement that variables be annotated with the locks protecting them. A number of recent examples, such as the Linux kernel developers' adoption of explicit user/kernel pointer annotations, support the practicality of annotations.

In our experience with Autolocker, the cost of annotating code was often offset by the savings of not explicitly acquiring locks. Generally, we found that lock annotations "scale better" than manual locking in the sense that very complex locking policies are specified much more easily using annotations. For less complex policies, using annotations usually required somewhat more effort.

A more worrisome problem of depending on user annotations is that they might be incorrect. Users can make mistakes when they write protection annotations, and these mistakes may lead to race conditions. Put another way, Autolocker guarantees the absence of races *only with respect to the annotations it is given*. Programs compiled using Autolocker never access a memory location without holding the lock protecting it, but they may race to access a location that is shared but not annotated as such. However, we stress that a race is only possible when a memory location has no protection annotation at all; it is not possible to create races by writing "incorrect" annotations. In the future, we plan to eliminate races entirely by using a conservative escape analysis (e.g., [30]) to find shared data that lacks protection annotations.

Deadlocks. A second Autolocker caveat is that it may reject programs if it is unable to order locks in a way that guarantees freedom from deadlocks. In AOLserver, our largest evaluation, Autolocker rejected 4 of 82 modules. Rejections are typically due to conservatism in Autolocker's alias analysis. It is always possible to solve the problem by replacing the offending locks with global ones, since they are easy to analyze. We give more details on this prob-

lem in Section 6, including a proposed improvement to the analysis that should accept the 4 previously rejected modules.

Two-phase locking. In our experiments, we found that Autolocker is capable of expressing a wide range of synchronization policies. However, it has several limitations built into it. The most important design limitation is that Autolocker implements only two-phase locking policies: once a lock has been released, no new locks can be acquired. This model is very common in databases because it guarantees serializability of transactions (the atomicity, consistency, and isolation properties of the ACID model [29]).

Unfortunately, some policies don't fit this model. A typical one is tree traversal using lock pairing. When walking down a path in a binary tree, the lock on a child node is acquired and then the lock on the parent is released. This policy is not two-phase because locks are acquired and released at every stage of the traversal. Fortunately, policies like this seem to be present only in the most complex systems like databases.

Other primitives. At the lowest level, Autolocker performs synchronization using mutexes, condition variables, and reader/writer locks (see Section 5). These primitives are sufficient, but not always optimal. For example, it is very natural to implement queues using semaphores. In Autolocker, a queue must instead use an alternate implementation based on locks, such as the efficient one presented by Michael and Scott [23], used in our experiments.

Shape information. Besides locking policy, Autolocker is also limited in its ability to understand data structure shape, which is a factor in deadlock detection. Consider the (somewhat artificial) case of a binary tree in which each node is protected by a separate mutex. The programmer would like to traverse down a path of the tree, acquiring locks for nodes along the way without releasing them. This example is deadlock-free because every thread will acquire locks in a given order: the lock for a node will always be acquired before the lock for a descendant. However, this lock order assumes that the data structure is actually a tree. In the presence of cycles, there is a possibility that the algorithm might deadlock.

Boyapati's race-free, deadlock-free Java [3] supports such tree structures, but requires a significantly more complex type system and compile-time analysis (to ensure that trees remain trees). Currently, Autolocker uses a very conservative alias analysis to distinguish memory locations and ensure there are no lock-order cycles. As a result, we may falsely detect a deadlock. In a later section, we describe a proposal by which the user can, in some cases, disambiguate memory locations using explicit memory regions. However, we have not implemented this technique.

2.2 Comparison with Optimistic Methods

We are not the first to recognize the problems of manual locking. The first proposal to use transactions in programming languages was by Lomet [22]; transactions ("actions") were implemented in the ARGUS distributed systems programming language [21]. More recently, researchers have proposed *optimistic* implementations of atomic sections, relying either on transactional memory libraries [12, 16, 17, 19] or on experimental transactional hardware [2, 24, 28]. Unlike lock-based systems, which handle data contention via waiting, these techniques roll back and repeat an atomic section each time contention is detected.

A central motivation for Autolocker is to explore a design that offers the ease of atomic sections while still relying on a pessimistic concurrency control policy. We wondered which benefits of transactional memory are attributable to atomic sections and which are attributable to optimistic concurrency control. Studies in the database community on concurrency control determined that neither an optimistic nor a pessimistic approach is strictly better, but led to the general belief that pessimistic locking works better

Approach	Programmer effort	Restrictions	Performance	Compatibility	Guarantees
Manual	acquire statements	none	total control	—	—
Optimistic	atomic blocks	poor I/O support	contention managers	can use locks in atomic {}	no deadlocks/no races
Pessimistic	atomic blocks & lock protections	deadlock check may fail	granularity control	existing locks handled seamlessly	no deadlocks/no protection violations

Table 1. A comparison of manual locking, optimistic atomic sections, and pessimistic atomic sections.

Lvals	q	::=	$x \mid q.f$
Exprs	e	::=	$n \mid q \mid e_1 \text{ op } e_2$
Stmts	s	::=	$q := e \mid \text{skip} \mid s; s \mid s \parallel s \mid \text{repeat } s \mid$
			$\text{assume } e \mid \text{begin}^1 \mid \text{end}^1 \mid \text{acq } q^2$
Types	τ	::=	$\text{int}[P] \mid \{ f_1 : \tau_1, \ldots, f_n : \tau_n \} [P] \mid \text{lock}_i$

P	$\in$	$\text{LockProtections} = \text{LockNames} \cup \{\bot\}$
i	$\in$	LockIDs

Figure 2. The grammar of our languages. Statements marked [1] are in both in AtomicC and LockC, but their meaning differs between the two languages. Statements marked [2] are only in LockC.

for systems with limited resources [1]. Table 1 shows a breakdown of the differences between manual synchronization and optimistic and pessimistic concurrency control.

Perhaps the most serious drawback of optimistic atomic sections is their use of rollback. In many cases, rolling back arbitrary program actions is not possible. A system performing network I/O inside an atomic section cannot unsend a packet. Several solutions to this problem based on buffering and customized compensation actions have been proposed, but they are not general enough to handle the entire range of program behavior [15]. Also, they require compensation or buffering code for functionality that commits side effects outside of memory. Pessimistic atomic sections, which do not use rollback, are immune to this problem.

Optimistic atomic sections also offer little control over performance aside from managing conflict resolution. A simplistic explanation of optimistic concurrency control is that it provides a maximally fine-grained sharing policy at the cost of high overhead, at least for the software-based implementations. Unfortunately, this trade-off is poor when concurrency is low, or when atomic sections are long or block, since longer sections increase conflicts.

Nevertheless, optimistic concurrency control can be quite desirable when locking offers unappealing solutions. For example, current fine-grained locking techniques for red-black tree data structures are very complex. Autolocker fails to support them because they are not two-phase and because they require shape information. However, transactional memory systems handle red-black trees easily and efficiently. Most tree operations access a small amount of data, so the copying and rollback overhead of the optimistic scheme is not dominant.

3. Language

We formalize Autolocker as a source-to-source transformation. The source language, called AtomicC, implements concurrency using atomic sections. The target language, LockC, also includes atomic sections, but it exposes the acquisition of locks that protect shared data. Figure 2 shows the grammar of both languages. The begin and end statements open and close an atomic section. In LockC, the acq (acquire) statement acquires a lock, which is released when the outermost atomic section ends.

Threads are not modeled directly in AtomicC or LockC. Each thread is modeled as a separate program. This simplification is possible because the properties we are interested in—deadlock

freedom and respect for lock protections—can be checked one thread at a time. A program is free from deadlocks as long as every thread acquires locks in a given order (although the order must be the same for all threads). A program respects lock protections as long as every thread individually respects them.

Aside from synchronization, these languages are extremely simple. We require that input programs use if and while statements for control-flow, but desugar them into an "assume e" statement combined with a non-deterministic choice ("$s_1 \parallel s_2$") or a non-deterministic loop ("repeat s") to simplify the type system and acquisition placement algorithm. The statement "if e then s_1 else s_2" is equivalent to "assume $e; s_1 \parallel$ assume $\neg e; s_2$". The statement "while e do s" is equivalent to "repeat (assume $e; s$)".

The set of values includes integers, locks, and records. Records are mutable. They are represented as in ML, where a record value is a reference to a heap structure that stores the field values. The type of a record includes a lock protection on the record reference. Our specification does not include variable declarations; it is assumed that the set of variables and their types is known. Functions are omitted for simplicity. In Section 5, we describe how the actual Autolocker tool handles the full set of C features.

Types in these languages include lock protection annotations in brackets. Every memory location (variable or record field) has an optional lock to protect it. These annotations are like the `protected_by` annotations used in the introduction, although they do not transitively extend to all reachable data. It is invalid to access a location without first acquiring the lock that protects it. Locks are described using *lock names*, which have somewhat unusual scoping rules. A lock name may refer to a global lock, but it may also reference fields inside of records. For example, if the variable x has type $\{ m : \text{lock}_1, f : \text{int}[m] \} [\bot]$, then $x.f$ is protected by $x.m$ (x itself is unprotected). Lock annotations like this are very important, since using only global locks would force programmers to use very coarse-grained locking policies. The simplest lock name is $\bot$, which means that there is no protection.

Since records are actually references, a lock protection can be attached to both the reference and to a field itself. If a variable x has type $\{ m : \text{lock}_1, f : \text{int}[m] \} [L]$, where L is some global lock, then accessing $x.f$ requires the lock L to dereference x as well as $x.m$ to access the field.

In order to keep track of locks in a program, every lock type has an identifier attached to it, denoted by a subscript. These IDs are automatically inserted by our Autolocker implementation. During execution, many different lock values can share the same ID.

3.1 Type System

This section presents a type system for AtomicC and LockC. The type system has two purposes. First, it ensures that integer and record values are never mixed up by the programmer. Second, it generates *computation histories* that summarize the synchronization-related behavior of a program. For an AtomicC program, a computation history guides how Autolocker places locks to generate a LockC program. The computation history for a LockC program is used to check if the program might deadlock or access data without the proper lock. A computation history without such violations is called *well-formed*. Autolocker is designed

Lvalues and Expressions $(S = \text{accumulated locks})$

$$(\text{VAR})\quad \frac{\Gamma(x) = \tau \qquad S = \emptyset}{\Gamma \vdash_{lv} x : \tau; S}$$

$$(\text{FIELD})\quad \frac{\Gamma \vdash_e q : \{\ldots, f_j : \tau_j, \ldots\}[P]; S}{\Gamma \vdash_{lv} q.f_i : \tau_i[\ldots, f_j \mapsto q.f_j, \ldots]; S}$$

$$(\text{LVALUE})\quad \frac{\Gamma \vdash_{lv} q : \tau; S}{\Gamma \vdash_e q : \tau; S \cup \text{locks}(\tau)}$$

$$(\text{OP})\quad \frac{\Gamma \vdash_e e_1 : \text{int}[L_1]; S_1 \qquad \Gamma \vdash_e e_2 : \text{int}[L_2]; S_2}{\Gamma \vdash_e e_1 \text{ op } e_2 : \text{int}[\bot]; S_1 \cup S_2}$$

Statements $(H = \text{computation history})$

$$(\text{REPEAT})\quad \frac{\Gamma \vdash_s s : H}{\Gamma \vdash_s \text{repeat } s : H^*}$$

$$(\text{CHOICE})\quad \frac{\Gamma \vdash_s s_1 : H_1 \qquad \Gamma \vdash_s s_2 : H_2}{\Gamma \vdash_s s_1 \ \| \ s_2 : (H_1 | H_2)}$$

$$(\text{SEQ})\quad \frac{\Gamma \vdash_s s_1 : H_1 \qquad \Gamma \vdash_s s_2 : H_2}{\Gamma \vdash_s s_1; s_2 : (H_1 \cdot H_2)}$$

$$(\text{ACQ})\quad \frac{\Gamma \vdash_e q : \text{lock}; \emptyset}{\Gamma \vdash_s \text{acq } q : \overset{\oplus}{q}}$$

$$(\text{ASSN})\quad \frac{\begin{array}{c}\Gamma \vdash_e q : \tau_1; S_1 \qquad \Gamma \vdash_e e : \tau_2; S_2 \\ \vdash \tau_2 \leq \tau_1 \qquad S = (S_1 \cup S_2) - \{\bot\} \qquad |S| \leq 1 \\ H = \text{str}(S) \cdot \text{str}(\{\overset{\oslash}{q_2} : q_2 \in \text{killed}(q)\})\end{array}}{\Gamma \vdash_s q := e : H}$$

$$(\text{SUBTYPE1})\quad \frac{}{\{f_1 : \tau_1, \ldots, f_n : \tau_n\}[P_1] \leq \{f_1 : \tau_1, \ldots, f_n : \tau_n\}[P_2]}$$

$$(\text{SUBTYPE2})\quad \frac{}{\vdash \text{int}[P_1] \leq \text{int}[P_2]}$$

Definitions

$$\text{locks}(\text{int}[P]) = P \qquad \text{locks}(\{f_1 : \tau_1, \ldots, f_n : \tau_n\}[P]) = P$$
$$\text{locks}(\text{lock}_i) = \emptyset \qquad\qquad \text{str}(\{q_1, \ldots, q_n\}) = q_1 \cdots q_n \qquad \text{(arbitrarily ordered)}$$

Well-Formedness

$$\text{WF}(H, <, \Gamma) \equiv \text{NoProtectionErrors}(H) \wedge \text{NoDeadlocks}(H, <, \Gamma)$$

$$\text{NoProtectionErrors}(H) \equiv \forall q. \ L(H) \cap L(([\overset{\oplus}{q}]^* q-) \mid (-\overset{\oplus}{q} - \overset{\oslash}{q} - q-)) = \emptyset$$

$$\text{NoDeadlocks}(H, <, \Gamma) \equiv \forall q, q_1, q_2. \ \text{id}(\Gamma, q_1) \leq \text{id}(\Gamma, q_2) \wedge q_1 \neq q_2 \Rightarrow L(H) \cap L(([\overset{\oplus}{q_1}]^* \overset{\oplus}{q_2} - \overset{\oplus}{q_1}-) \mid (-\overset{\oplus}{q} - \overset{\oslash}{q} - \overset{\oplus}{q}-)) = \emptyset$$

Figure 3. An excerpt of the type system for AtomicC and LockC. The rule for assume is not shown, but is similar to an assignment.

---

so that the LockC program it generates always has a well-formed computation history (as we will prove). Thus, the type system is useful both as an algorithmic tool for transforming AtomicC programs into LockC programs and also as a formal tool to ensure that Autolocker works correctly.

Histories. The computation history of a statement is a regular expression that summarizes the effect of the statement. Every path through a statement is represented by the history. Histories give our analysis a measure of path sensitivity while still being linear in the program size. The alphabet of a computation history has three symbols for every lvalue q:

- $\overset{\oplus}{q}$ means a lock with lvalue q is acquired.
- q means a value protected by lock q is accessed.
- $\overset{\oslash}{q}$ means a lock with lvalue q is killed by an assignment.

A lock lvalue is *killed* whenever the lock it refers to might change. This occurs when a programmer updates a location which can alias q. For instance, the common case of assigning to a record variable r whose type contains a lock field lk would kill lock lvalue $r.lk$.

For notational convenience, we use a dash $(-)$ instead of $.^*$ to represent an arbitrarily long string of symbols. For an example history, consider the following program.
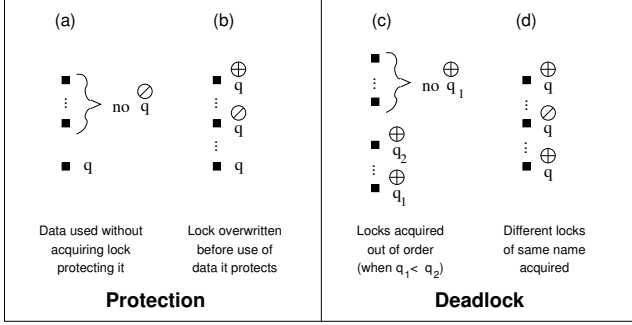
```
x, z : { L : lock, v : int[L] } [⊥]
acq x.L;
y := x.v + 1;
(x := z) ∥ skip
```

Let q be the lvalue $x.L$. This program generates the computation history $\overset{\oplus}{q} q (\overset{\oslash}{q} | \epsilon)$.

The type system is shown in Figure 3. Notice that it has no rules for begin and end statements. To simplify the presentation, we assume that each atomic section is checked separately. Each section will generate one computation history; all histories must be well-formed in order for the full program to be well-formed. Our implementation uses an inter-procedural analysis to extract atomic sections from code before generating computation histories.

Locks. The typing rules for expressions and lvalues determine the set S of lock lvalues that must be held for correct execution. When

Figure 4 region content:

(a)   (b)   (c)   (d)

no $\oslash\atop q$ $\oplus\atop q$ no $\oplus\atop q_1$ $\oplus\atop q$
              $\oslash\atop q$ $\oslash\atop q$
                                $\oplus\atop q_2$
              $\oplus\atop q$ $\oplus\atop q_1$ $\oplus\atop q$

Data used without | Lock overwritten | Locks acquired | Different locks
acquiring lock | before use of | out of order | of same name
protecting it | data it protects | (when $q_1 < q_2$) | acquired

Protection | **Deadlock**

Figure 4. A demonstration of violations of well-formedness rules. The formal definition of the well-formedness rules is in Figure 3.

a lock is embedded inside a structure, there is a slight complication. Consider the following example:

$$x : \{\, L : \text{lock}, v : \text{int}[L] \,\} \, [\bot]$$
$$y := x.v + 1$$

The (FIELD) rule is used to check $x.v$. It has type $\text{int}[L]$. However, the typing rule substitutes $x.L$ for L, so that the final type is $\text{int}[x.L]$. We use syntactic checks on types (not shown here) to ensure that field names never shadow each other; otherwise the substitution could be confused.

The rules for expressions accumulate the set of required locks in the variable S. The (LVALUE) rule uses the type $\text{int}[x.L]$ to determine that the lock $x.L$ is needed to evaluate this expression.

Assignments. Assignments are handled in a mostly standard way. We use a subtyping judgment to determine whether it is valid to assign one type to another. This judgment forbids assigning locks and ignores top-level lock protections since they are not changed by the assignment.

To simplify the formal presentation of Autolocker, an assignment statement is allowed to access locations protected by at most one lock. It is easy to transform a program into one that meets this requirement by adding unprotected temporary variables. Additionally, we forbid a lock lvalue acquired in an acq statement to need any other locks in order to be evaluated. The reasons for these restrictions will become clear when we discuss acquisition placement.

Tracking locks in the type system via their lvalues causes some difficulty, since lvalues can refer to different concrete memory locations as the heap is updated. To preserve safety, the (ASSN) rule records in the history all the lvalues q whose value might be affected by an assignment, as defined by the killed function. Although this kill set may theoretically be infinite, the only elements that matter are lock lvalues that appear elsewhere in the computation history, which will certainly be finite. The kill set can be computed by any sound alias analysis. The Autolocker implementation uses a very conservative analysis that assumes that any two non-global locks may be aliased.

Well-formedness. There are four well-formedness conditions. Figure 4 (a-d) show how the well-formedness rules can be violated by a computation history. The conditions are written formally in Figure 3.

The first two well-formedness conditions, (a) and (b), are violated if data is accessed without acquiring the lock protecting it. The first condition is the most straightforward. A symbol q somewhere in the history means that a memory access protected by lock q occurs at that point. If the access is not preceded by an acquisition of the lock (represented by $\overset{\oplus}{q}$ in the history), then condition (a) is violated.

The second condition closes the loophole where a lock becomes inaccessible (due to an assignment) between being acquired and being needed. Consider this LockC code:

$$x : \{\, L : \text{lock}, v : \text{int}[L] \,\} \, [\bot]$$
$$\text{acq } x.L;$$
$$x := \cdots ;$$
$$x.v := x.v + 1$$

When $x.v$ is accessed, the lock that protects it at that point may not be held. The history for this code is $\overset{\oplus}{q}\overset{\oslash}{q}q$, where $q = x.L$. The second well-formedness condition, (b), recognizes histories like this one and forbids them.

The latter two conditions, (c) and (d), guarantee that the program contains no deadlocks. They ensure that locks are acquired in a global, *a priori* order that holds for the entire program. Locks are ordered by a relation $<$ on their IDs. In this section, we assume the order is given up front; the next section describes a process for computing a valid ordering. The rules, given formally in Figure 3, assume a function $\text{id}(\Gamma, q)$ has been defined that uses the typing rules to determine the ID of a given lock lvalue. Occasionally, we write $q_1 <_\Gamma q_2$ to mean that the IDs of q_1 and q_2 are related by $<$.

Condition (c) checks for deadlocks involving two distinct locks q_1 and q_2 whose IDs are related by $\leq$. It simply states that it is illegal to acquire q_2 before q_1, since this sequence would violate the given lock order. It does not prevent q_2 from being acquiring in between two acquisitions of q_1, since the second time q_1 is acquired is a no-op according to our semantics. However, it does prevent two distinct lock lvalues with the same ID from being acquired, since $<$ is not reflexive (a reflexive relation is cyclic and might permit deadlocks).

Condition (d) closes a loophole in (c) due to assignments to locks, much as (b) does. Consider the following code.

$$x : \{\, L : \text{lock}, v : \text{int}[L] \,\} \, [\bot]$$
$$\text{acq } x.L;$$
$$x := \cdots ;$$
$$\text{acq } x.L$$

This code in fact acquires two potentially different locks that have the same lvalue. Condition (c) is not violated in this case, because both acquisitions use the same lvalue. Thus, we use case (d) to forbid situations where a lock is acquired twice with an assignment in between that might change its value.

3.2 Soundness

Computation histories make the type system for LockC appear slightly exotic. It is not at all obvious that the four well-formedness conditions prevent every possible deadlock or lock protection violation. To gain assurance that the conditions work, we formulated an operational semantics for LockC that is more conventional. In this semantics, the state of a program includes the set of held locks. It is illegal to access a memory location without holding the lock protecting it, or to acquire a lock when one ordered after it is already held. We proved that programs that pass our type system will never go wrong according to this semantics. The proof is summarized in Appendix A.

4. Acquisition Placement

The main purpose of Autolocker is to place lock acquisition statements throughout a program in a way that ensures race and deadlock freedom. The lock placement algorithm described in this section has two stages.

1. First, it determines an order in which locks should be acquired for a given AtomicC program. It does so by discovering data dependencies between locks. These dependencies form a partial

order on the lock IDs. If this order is cyclic, then the program is rejected. Otherwise, the partial order is converted to a total order using a topological sort.

2. In the second stage, lock acquisitions are placed throughout the program in the order inferred previously. The placement algorithm guarantees that any lock required by a statement will be acquired before the statement executes and that locks are always acquired in the order determined in the first stage. The output of this stage is a LockC program.

In the rest of this section, we describe the two phases of the algorithm in greater detail. We then prove that the placement algorithm always generates type-correct programs that pass the well-formedness checks.

4.1 Order Inference

The goal of lock order inference is to determine a minimal set of constraints on the order in which locks must be acquired. Each new constraint increases the likelihood that the input program will be rejected, so the set should be as small as possible. The only constraints that are absolutely necessary are those caused by data dependencies, such as the one in the following example:

$$G : \text{lock}; x : \text{int}[G]; y : \{ L : \text{lock}, v : \text{int}[L] \}$$
$$x := x + 1;$$
$$y := \cdots;$$
$$x := y.v$$

The first statement in this example requires that the lock G be held. The third statement requires that $y.L$ be held. It is impossible to acquire $y.L$ before G, because the value of y is not even available at the time when G must be acquired. (Of course, an intelligent tool might be able to reorder the assignment. However, in real C programs pointers make this kind of transformation so difficult as to be pointless.)

Intuitively, the main goal of the order inference algorithm is to determine when one lock acquisition can be moved before another. Data dependencies, such as the one above, make this movement impossible. The acquisition of a lock lvalue can be moved earlier in the program as long as it does not cross any assignments that affect the lvalue. Thus, assignments in the program act as potential barriers over which lock acquisitions cannot be moved.

The use of computation histories makes it fairly easy to determine a lock order. Imagine a computation history $q_1 \overset{\oslash}{q_2} q_2$. In this history, lock q_1 must be acquired before q_2, since q_2 is (potentially) assigned after q_1 must already have been acquired. Even if q_2 were acquired before q_1, the assignment means that the lock required later on might be different than the one that was acquired.

Thus, the first step of our algorithm uses the typing rules of the previous section to generate computation histories for each atomic section in a program. Let H be the set of all these histories. Then we define the dependencies as follows:

$$\text{deps}_0(H) = \{(q_1, q_2) : L(H) \cap L(-q_1 - \overset{\oslash}{q_2} - q_2-) \neq \emptyset\}$$

These dependencies are defined in terms of lock lvalues. However, the placement algorithm in the next section knows only about lock IDs. We convert $\text{deps}_0(H)$ to a relation on IDs, $\text{deps}(H)$, by mapping lvalues to their IDs using the environment Γ. This relation, in turn, forms a directed graph. If this graph is cyclic, then there is no valid ordering of lock acquisitions and the program is rejected due to a possible deadlock. Otherwise, we use topological sort to generate a total order $<$ on the locks IDs. This order serves as input to the next stage of the algorithm.

$$\frac{\mathcal{T}[\![s_2 \text{ with } L]\!] = s_2' \text{ with } L' \qquad \mathcal{T}[\![s_1 \text{ with } L']\!] = s_1' \text{ with } L''}{\mathcal{T}[\![s_1; s_2 \text{ with } L]\!] = s_1'; s_2' \text{ with } L''}$$

$$\frac{\mathcal{T}[\![s_1 \text{ with } L]\!] = s_1' \text{ with } L_1 \qquad \mathcal{T}[\![s_2 \text{ with } L]\!] = s_2' \text{ with } L_2}{\mathcal{T}[\![s_1 \parallel s_2 \text{ with } L]\!] = s_1' \parallel s_2' \text{ with } L_1 \cup L_2}$$

$$\frac{\begin{array}{c} L' = L \cup \text{locks}(q := e) \\ A = \{q \in L' : \exists q' \in \text{locks}(q := e).\, q \leq_\Gamma q'\} \\ s = \text{acquires}(A) \end{array}}{\mathcal{T}[\![q := e \text{ with } L]\!] = (s; q := e) \text{ with } L'}$$

Figure 5. The algorithm to add lock acquisitions to an AtomicC program based on a given lock ordering. The acquires$(\cdot)$ function returns a sequence of ordered acquire statements given a set of locks. The locks$(\cdot)$ function uses the typing rules to determine all the locks that are required for a given statement to execute.

4.2 Acquisition Placement

The lock placement algorithm inserts lock acquisitions before statements that need them. Only assignments and assume statements require locks. We use the typing rules to determine the set S of locks needed by expressions in a statement. For any statement s, call this set locks(s).

However, adding only necessary lock acquisitions may order locks improperly. Imagine a program with the history $q_2\, q_1$. Naively acquiring locks only when needed would result in the history $\overset{\oplus}{q_2}\, q_2\, \overset{\oplus}{q_1}\, q_1$. However, this history is not well-formed if $q_1 <_\Gamma q_2$ (the locks are out of order). In that case, we must "preventively" acquire q_1 first, resulting in the history $\overset{\oplus}{q_1}\, \overset{\oplus}{q_2}\, q_2\, \overset{\oplus}{q_1}\, q_1$. The second acquisition of q_1 is unnecessary, but it can be eliminated later as an optimization.

The algorithm $\mathcal{T}[\![s \text{ with } L]\!]$ for adding lock acquisitions to a statement s is shown in Figure 5. It takes as input the set L of locks that may be acquired after s, used to do preventive acquisitions. It returns an updated version of this set that includes the locks needed by s.

The rules for sequencing and choice are unremarkable. They simply thread the L set through the statements in a syntax-directed manner. The rule for assignments is more interesting. In it, the set L' includes locks needed by the assignment as well as any other locks acquired later on, from L. The locks to be acquired for the assignment are placed in A. A lock is in A if it is in L' (i.e., if it is needed by the assignment or by a later statement), and if it is $\leq$ some lock needed by the assignment. This includes both the locks used by the assignment as well as the locks to be acquired preventively. A sequence of acquire statements is synthesized from A using the acquire$(\cdot)$ function (definition not shown). When applied to a set of lock lvalues, it returns acquire statements ordered according to the lock ordering.

4.3 Correctness

The transformation $\mathcal{T}$ should always generate a well-typed program with a well-formed history that is semantically equivalent to the original program. Semantic equivalence is obvious, since only acq statements are added, and they have no observable effect on the program aside from synchronization.

Proving that the transformed program is well typed is slightly more difficult. We must show that every additional acq statement is well typed. This amounts to proving that lock lvalues being acquired do not need any other locks in order to be evaluated, as required by the (ACQ) typing rule. However, assignment state-

ments, which result in lock acquisitions being added by $\mathcal{T}$, need at most one lock (the $|S| \leq 1$ restriction). Observe that if this lock depended on other locks, then the assignment would also require those locks. Since it doesn't, acquire statements depend on no locks, and so they are well typed.

Lastly, we prove that $\mathcal{T}$ always generates LockC programs with well-formed histories. It is not obvious that this is so. For example, it is not clear that the dependencies inferred by deps_0 are sufficient, or that $\mathcal{T}$ acquires the right locks at the right times. In this section, we show why $\mathcal{T}$ works. First, we need a lemma that describes the way that histories behave under transformation.

LEMMA 1 (Histories). *Let H be the computation history of a program s, let $<$ be the lock order inferred for s, and let H' be the history of the transformed program $\mathcal{T}[\![s \text{ with } \emptyset]\!]$. Then the following properties hold.*

*(i) If there is a history $h' \in L(H')$ such that $h' \in L(-q_1 - \overset{\oslash}{q_2} - q_2-)$, then $q_1 <_\Gamma q_2$. That is, if there is a dependency in a history in H', then this dependency was discovered by the order inference algorithm.*

(ii) If a history $h' = h_0 \overset{\oplus}{q_1} h_1 \in L(H')$, then $h_1 = h_2 q_2 h_3$, where h_2 contains no kills and $q_1 \leq_\Gamma q_2$. In other words, every lock acquisition is eventually followed by a lock use ordered after it, and no kills occur in between.

(iii) If there is a history $h' = h_0 \overset{\oplus}{q} h_1 \in L(H')$, then there is also a history $h'' = h_0 \overset{\oplus}{q} h_2 q h_3 \in L(H')$. That is, if a history contains a lock acquisition, then there is another (not necessarily distinct) history that is identical to the first up to the acquisition, but then later uses the lock that was acquired.

In brief, part (i) is true because the transformed history differs from the original one only in acquisitions. Part (ii) is true based on how $\mathcal{T}$ adds acquisitions—they are always added right before a required lock, and only locks that are $\leq$ some required lock are added. Part (iii) is true because $\mathcal{T}$ adds an acquisition only if that lock is needed some time later (although it may be needed on a different path).

Now we use this lemma to show that the history of the transformed program $\mathcal{T}[\![s \text{ with } \emptyset]\!]$ satisfies well-formedness properties (a-d), and thus that the transformed program is free of deadlocks and data races.

WF(a) — No memory access is unprotected. The algorithm above inserts a lock acquisition for every lock needed by the original history. Based on the (ACQ) typing rule, the acquisitions it adds do not result in any other lock requirements, so every memory access in the new history is covered.

WF(b) — Killed locks are never re-acquired. Consider this illustration.
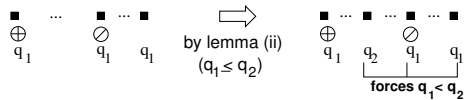
If condition (b) is violated, then there must be a history in H' like the one on the left. Then by part (ii) of the lemma, the initial acquisition of q_1 must be followed by a use of some q_2, where $q_1 \leq_\Gamma q_2$. But then this history (shown on the right) contains a data dependency between q_2 and q_1. By part (i) of the history lemma, this dependency also existed in H, so it must have been discovered by the order inference algorithm. The algorithm would have forced $q_2 <_\Gamma q_1$. However, the two inequalities contradict each other, so this can never happen.

WF(c) — Locks are acquired in order. This property is guaranteed by $\mathcal{T}$. If two locks q_2 and q_1 are acquired out of order, then the earlier acquisition of q_2 must have been added when q_1 was in the set L. But then q_1 should have been placed in the A set along with q_2. The acquires$(\cdot)$ function is defined to acquire locks in order. Thus, q_1 was actually acquired first, so this condition cannot be violated.

WF(d) — Protecting locks are never killed after acquisition. Assume that this occurs. Then there must be a history in H' like the left one below.
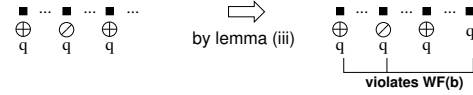
However, by part (iii) of the history lemma, the final acquisition must be followed by a use in some other history in H'. This history will violate well-formedness condition (b), which we have proved will not happen.

5. Extensions

Real programs are built using a variety of features that are inexpressible in AtomicC. In the next few sections, we describe some extensions we implemented that were required for Autolocker to be useful in practice.

5.1 Condition Variables

Condition variables are common in programs that use the standard C libpthread library. There are two operations. A thread can *wait* on a condition variable until another thread *signals* the variable. Typically, condition variables are used to wait until a predicate on the program state becomes true. Any thread that updates a variable that might affect the predicate must signal the condition variable. One problem with this approach is that it is easy to forget to signal a condition variable in every circumstance.

Autolocker treats condition variables in much the same way as locks. A variable x may be annotated with a condition variable c. Whenever x is updated, Autolocker automatically signals c. Programs using Autolocker can wait on predicates (C expressions). Autolocker infers the condition variable associated with the predicate (by examining the variables that are subexpressions) and waits on it. The use of Autolocker eliminates errors where a variable is updated but the corresponding condition variable is not signaled. This provides similar functionality to Harris and Fraser's atomic statement guards [16]. However, Autolocker is somewhat weaker since libpthread allows the programmer to wait on only one condition variable at a time. Also we do not yet provide a way to choose whether to signal one or all waiting threads.

5.2 Prelocking

AtomicC requires that every variable be explicitly annotated with the lock that protects it. However, there are times when no single lock protects a variable. For example, a program may contain two hash tables, each protected by a different lock. Depending on which path is taken, the programmer may assign a local variable to point to one hash table or the other. Autolocker includes a special protection annotation written $locked. A variable marked $locked (a *prelocked variable*) may contain data that is protected by any arbitrary lock as long as the lock is held throughout the lifetime of the variable.

Autolocker is responsible for acquiring the correct lock whenever an assignment is made to a $locked variable, as follows. The full type system includes a subsumption rule so that data protected by a lock can be converted to $locked data. Each coercion generates code to acquire the lock being coerced in order to maintain the

invariant of the prelocked variable. Additionally, prelocked variables must be well nested with respect to atomic sections: an atomic section cannot end during the lifetime of a `$locked` variable. In particular, global variables cannot be prelocked.

Variables marked `$locked` are particularly useful for function parameters, since Autolocker does not have any parametric polymorphism. A function with a prelocked parameter can be called with either a protected or an unprotected value. If the value is protected, the lock protecting it is acquired before the function call. This approach has the advantage that the callee may be completely ignorant of locking. This property is important for supporting external libraries.

5.3 Reader/Writer Locks

Reader/writer locks extend mutexes by allowing a lock to be acquired by *n readers* or by 1 *writer*. They can significantly increase available concurrency for programs accessing mostly read-only data structures. For instance, a hash table protected by a global read/write lock might allow concurrent lookups.

Autolocker supports using either mutexes or reader/writer locks on a per-lock declaration basis. If a given atomic section includes a write to something protected by a reader/writer lock L, then all acquires of that lock in that atomic section will be write acquires. Otherwise, the acquires will be read acquires (acquiring a lock first for read and then for write can lead to deadlock, from ch 17.3 of [29]).

5.4 Multi-Granularity Locking

In many cases, reader/writer locks are used in a hierarchical fashion. For instance, similar to our first example, a hash table may have a global reader/writer lock protecting the hash table's arrays and mutexes protecting each bucket. If the program has acquired the global lock for writing (e.g., because it is resizing the table), it does not need to acquire the bucket locks. The bucket locks are used when accessing individual hash entries; in this case, the global lock is held for reading. This organization, called multi-granularity locking in the database literature [14], increases the concurrency of operations that only modify individual buckets.

In Autolocker, a lock L can be declared as *subordinated* to a reader/writer lock L'. Autolocker will automatically acquire L' whenever it would acquire L; additionally, if L' is known to be acquired for writes, the acquires of L will be suppressed. Note that this also implies that L' must come before L in the global lock order. Our current implementation only supports subordination to global locks. We expect to extend this to locks specified relative to L (e.g., `parent->lock`); we have not implemented this yet as it would require better alias analysis in Autolocker to be useful.

6. Evaluation

Autolocker is implemented as a 2,500 line extension to the CIL C analysis framework [25]. Although the algorithm has been described in previous sections via the intersection of regular languages, the actual Autolocker tool uses more conventional techniques. It analyzes lock usage and kill information using a flow-insensitive type analysis. It determines lock ordering via a dataflow analysis.

For call-graph construction, functions are grouped into equivalence classes as in a standard unification-based alias analysis. The call-graph is used to determine computation histories for the program. For simplicity, local lock lvalues are killed at the end of a function. In some cases, this simplification may lead the lock ordering algorithm to be overly conservative. In the rare case that it occurred, inlining solved the problem.

We unsoundly assume that external library functions acquire no locks. Eventually, we plan to offer a mechanism where lock ordering is specified in module interfaces. However, in our benchmarks, locking was not used by any external libraries. Autolocker *does* allow external libraries to access shared data. For this to work, the programmer must add `$locked` annotations to the library interface. We used this feature extensively in our larger benchmark.

The goal of the evaluation was to test whether Autolocker is efficient, practical, and expressive. To test efficiency, we used a highly concurrent hash table and a FIFO queue with less potential for concurrency. We compared Autolocker to manual locking and to two recent software transactional memory implementations. The Autolocker versions performed comparably to manual locking, scaled well and had significantly higher throughput (from 1.6x to 3.4x) than the transactional memory versions. For measuring the practicality and expressiveness of our tool, we annotated a large (50,000 line) open-source web server. We attempted to use exactly the same locking policy in the Autolocker version of the server as in the original, and we succeeded in 78 out of 82 modules. The rest of this section presents our experiences, performance data, and possible improvements to the Autolocker algorithm.

6.1 Hash Table Microbenchmark

We implemented a concurrent, non-resizable hash table using Autolocker and compared it with several other versions of the same data structure. These included a single-threaded version, manual locking versions using coarse- and fine-grained locking, two Autolocker versions with the same trade-offs, and two versions using software transactional memory. One STM used an object-based transaction manager by Fraser and Harris [13]. The other used a similar transaction manager by Ennals with better cache locality and a different way of handling conflicts [5].

We used a non-resizable hash table implementation because the transactional memory systems we benchmarked had limitations on object size and on the total amount of memory that can be touched in a single transaction.

We ran benchmarks on an Intel server with four 1.9 GHz Xeon processors with HyperThreading. Each processor had 512 KB of cache and the machine had 1 GB of main memory. The mix of hash table operations was 10% insertions, 10% deletions, and 80% lookups. We did not see significant relative variations when we varied the load. We used a hash table load factor of $\alpha = 1$, and the benchmarks seemed fairly insensitive to this parameter as well. We measured the number of operations per second performed by a single thread (top graph in Figure 6), as well as the total number of operations per second (bottom graph).

As would be expected, Figure 6 shows that the single-threaded version performs best in terms of operations per second *per thread*, since it has no contention or locking overhead at all. The Autolocker versions performed slightly worse than manual locking version for both coarse- and fine-grained policies. The difference is due to the overhead of the Autolocker runtime, which must track the set of locks owned by a thread at any time. The versions with coarse-grained policies performed slightly better than their fine-grained counterparts for one thread, but they are unable to take advantage of extra processors.

The fine-grained policies scaled fairly well up to four threads, which is the number of processors in the machine. In the Autolocker version, each thread of the 4 thread configuration performed only 30% less work than with only one thread. As the number of threads increases beyond four, each thread naturally must do less work. However, in the 32 thread configuration of Autolocker, each thread performed 144 operations per second, which is actually better throughput overall than the 4 thread version. The highest total throughput was achieved with 6 threads, which means that the fine-grained implementations took advantage of HyperThreading.
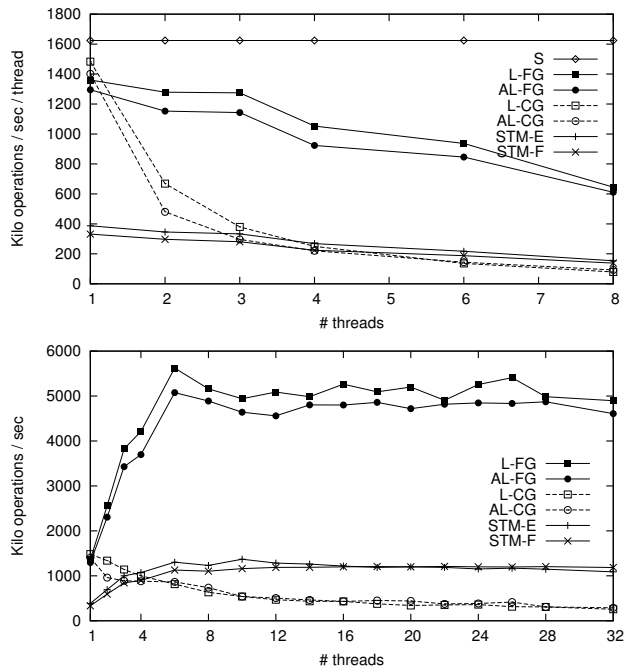
Figure 6. Hash table performance with 10% insertions, 10% deletions, load factor of 1. The top graph shows operations per second of a single thread for up to 8 threads. The bottom graph shows *total* operations per second for up to 32 threads.

The transactional memory versions had very high overhead, but also scaled well. In general, the transactional memory manager by Ennals [5] performed somewhat better than the one by Harris and Fraser [13]. The Ennals version performed 30% worse between the 1 and 4 thread versions—about the same scaling as Autolocker. The Ennals manager also scaled to 32 processor about as well as Autolocker did. However, overhead was clearly significant for both transaction managers. At 4 threads, the Ennals manager was 3.4x slower than Autolocker.

6.2 FIFO Microbenchmark

We did another microbenchmark with different FIFO queue implementations. The FIFO we implemented has infinite length and does not block waiting when it is empty. It is implemented as a singly linked list with head and tail pointers. We benchmarked versions that used either one lock to protect the entire structure or two locks to protect the head and the tail. The two-lock version is based on the algorithm from Michael and Scott [23]. We implemented these locking versions using manual locks and Autolocker. We also tested variants that used the transaction managers from Fraser et al. and Ennals.

The experiments were done on a server with four 2.0 Ghz Xeon CPUs. An equal number of producer and consumer threads contended for access to the queue. Figure 7 shows performance numbers. In general, throughput did not increase with more threads because the head and tail pointers were heavily contended, limiting concurrency to at most two threads. This is very different from the hash table experiment, where contention was low. The Autolocker versions were about 20% slower than the manual versions. In all cases, the two-lock versions performed better than STM versions. The Autolocker version with two locks was 1.6x to 2x faster than the transaction manager of Fraser et al.

Note that memory layout affected synchronization performance significantly. All the variants tested above used the default gcc
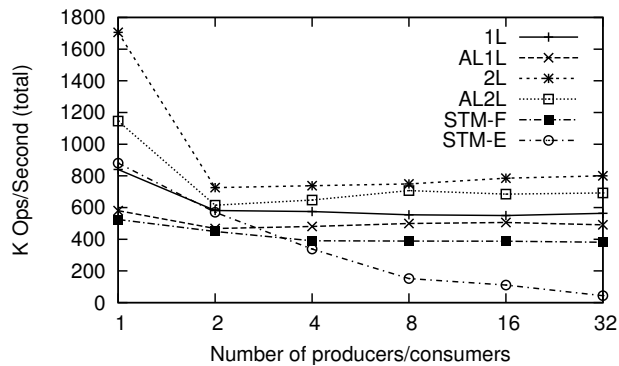
Figure 7. FIFO performance with different concurrency levels.

memory layout. Autolocker gives the programmer the freedom to lay out memory as desired, just as with manual locking. As an experiment, we aligned data in the two-lock Autolocker variant so that head pointer and its lock were on a different cache line from the tail pointer and its lock. This version performed about 40% better as seen in Figure 7. Transactional memory managers do not give the programmer complete freedom to control cache layout, since the transaction managers controls shared data.

6.3 AOLserver

AOLserver is an open-source web server initially released by America Online. We chose it as a benchmark because of its heavy use of threads as well as its reliance on a large external library (Tcl), which is challenging for whole-program analysis. We made several modifications to AOLserver to use it with Autolocker. First, we replaced each mutex acquisition/release with the beginning/end of an atomic section. Most of these (73% of 143) were lexically scoped. In a few cases where lock ownership crossed function boundaries, atomic sections were enlarged somewhat to encompass more code, since Autolocker forces all atomic sections to begin and end in the same function. We broadened fewer than 10 atomic sections.

Next, we examined the code to determine the variables that were shared and the locks that protected those variables. This task was inherently difficult, since the codebase was large and unfamiliar. We added lock protection annotations to each shared variable. Our goal was to use exactly the same locking policy as the original code. To check for errors, we preserved the original locking code inside each atomic section. If the Autolocker runtime did not acquire the same locks as the original code, an error was triggered. This technique does not guarantee that all lock protection annotations are correct, but it does ensure dynamically that the modified program has at most as many race conditions as the original one.[2]

Besides lock protections, we also added a number of condition variable and prelocking annotations. Prelocking was particularly useful when the AOLserver code made use of Tcl data structures (such as hash tables) that were not designed to be protected by locks. This forms an important modularity advantage for Autolocker: we can easily integrate third-party code that was not designed for concurrency (unlike software-based optimistic approaches).

---

[2] Interestingly, although Autolocker is not designed to detect race conditions, we found several likely races while studying the code. Explicitly writing lock protection annotations is a useful mechanism for formalizing and documenting the information that most programmers store only in their heads.

Autolocker also required 175 "trusted" casts, in which lock protection annotations were added or removed from a type. Many of these trusted casts were necessary in order to compensate for C's impoverished type system. Of the 175 trusted casts that we added to AOLserver, we estimate that all but 23 could be eliminated with a more expressive type system. In particular, 51 casts were caused by C's lack of parametric polymorphism, which was problematic when locked data was added to data structures. Forty-five casts were due to standard C library functions like `malloc` and `memcpy`, which operate on `void*` data. Twenty-four casts were used when dealing with callback functions, which could be solved with closures, objects, or existential types. Sixteen casts were used to approximate abstract types in C, and another sixteen were used to circumvent the type system in miscellaneous ways.

The remaining 23 of 175 trusted casts were needed when shared data became unshared, or vice versa. It is not uncommon for data to be shared only under certain conditions, such as when it is stored in a data structure. Autolocker's static type system does not adequately characterize such relationships. We used trusted casts to circumvent the problem, since our goal was to replicate AOLserver's original locking policy as closely as possible. However, a better solution might be to conservatively assume that the data in question is always shared. There may be a performance cost in additional locking overhead. However, we expect it to be small, since the locks only need to be acquired inside atomic sections; the data is unlikely to be accessed there during the time when it is unshared.

In four modules of AOLserver (out of a total of 82), we were unable to replicate the original locking policy. In these cases, the Autolocker analysis was too conservative and the program was rejected for deadlocks. To compensate, we coarsened the locking granularity so that the analysis would succeed. Each of the four modules created networks of dynamically allocated objects. The objects in a network were protected by a single lock, which was stored in a central object. The remaining objects kept a pointer to the central object in order to access the lock. Frequently, several objects in a network were accessed in a single atomic section. Autolocker rejected the programs because it was unable to determine that the objects being accessed all belonged to the same network, and thus used the same lock. If the objects had been of different networks, then the program might have deadlocked. The "same network" property is a data structure shape invariant that is difficult to infer statically.

We believe that this problem can be solved using regions. Traditionally, regions are used to group objects that have the same lifetime. The type system ties the liveness of each object to the liveness of the region itself, which is usually well-known statically. In the case of Autolocker, objects in a region would share the same lock. The lock would be tied to the region itself, rather than to any particular object. In order to ensure deadlock freedom, the type system would simply ensure that the objects being accessed in an atomic section belong to the same region, which is much more tractable than checking shape information. We believe adding regions will broaden the set of programs Autolocker accepts.

We benchmarked AOLserver using Apache bench on a two-processor machine with four machines to generate load. The load generators requested a single two-byte file in order to reduce I/O latency and make the CPU the bottleneck. The throughput penalty of Autolocker ranged from 2.0% to 4.9%. In a more realistic configuration, the performance penalty would likely disappear all together.

In total, about 1% of the AOLserver codebase was affected by the Autolocker transition. The most difficult and error-prone annotations to generate by far were those related to data sharing. However, their presence improves the program's documentation.

7. Related Work

Autolocker is complementary to dynamic race detection techniques, such as those based on locksets [26, 32], happens-before relationships [4, 33], or both [34]. These tools detect the absence of locks and thus could be used to infer missing lock annotations. However, their success depends on the specific execution and may have false negatives.

Flanagan and Freund [6] and Boyapati [3] use `guarded_by` annotations to ensure race-freedom of Java programs. Boyapati's SafeJava system can express somewhat more complex locking policies than Autolocker, in particular through its "ownership" concept (we believe we can add some of SafeJava's features to Autolocker). Subsequently, Flanagan, Freund, Qadeer have combined `guarded_by` annotations with explicitly declared atomicity constraints (similar to `atomic` statements). They use static [11] or dynamic [7] analysis to ensure that the locking policy chosen by the programmer actually enforces the declared atomicity constraints. They have also developed analyses to infer `guarded_by` annotations [8], atomicity constraints [10] and to insert missing synchronization operations [9]. This last system is closest to Autolocker: based on explicit lock protection annotations and atomicity requirements, locking calls are automatically inserted. Sasturkar et al. [31], combine object ownership with Flanagan et al's annotations to be able to describe more locking policies. Of the system's above, only Boyapati's addresses the issue of deadlock, based on statically verifying locking calls against a programmer-specified lock order.

Excepting [9], these systems rely on manually inserted locking calls, and thus provide little benefit in actually writing or modifying code. Manually locking has two potential advantages over Autolocker: greater performance, and expression of more general locking policies. For the former case, our benchmark's show that in practice Autolocker's overhead is relatively low. For the latter, we believe that if a lock-checking algorithm has enough information at an access to prove that necessary locks are held, then it also has enough information to insert lock acquisition calls, and that this is the better policy.

Herlihy and Moss [20] proposed the first transactional memory hardware based on extending cache-coherency mechanisms; Rajwar and Goodman [27] used similar cache-coherency-based hardware transaction support to execute lock-based programs optimistically. These systems only support bounded-size transactions, and do not allow for context switches or page faults, making them impractical for implementing transactions in a programming language. These limitations have been addressed in the more recent work by Ananian et al. [2], Moore et al. [24] and Rajwar et al. [28]. All three propose hardware transaction mechanisms that support unbounded-size, long-running transactions, at the expense of fairly complex hardware support. None of these proposals is available in current or near-future hardware.

Software transactional memory systems, such as those implemented by Harris and Fraser [12, 16, 17] and Herlihy et al. [19], run on existing hardware. However, they have fairly high overhead. Ennals [5] argues that this overhead is in part due to (unnecessary) obstruction-freedom guarantees.[3] Ennals's system is based on two-phase locking of accessed objects; however, unlike Autolocker, it must support rollback as it relies on aborting transactions that are found to deadlock at runtime. It is thus more akin to the optimistic concurrency models of the other hardware and software transactional memory systems, and has the same problems with non-reversible actions such as I/O. Conversely, supporting rollback allows for more elegant and composable code within atomic sections, as proposed by Harris et al. [18].

---

[3] A system is obstruction-free if suspending one thread does not prevents others from making progress.

The relative merits of optimistic and pessimistic concurrency control were the subject of many studies in the database literature [1]. For that community, the collective wisdom appears to be that optimistic is only better under excess resources (due to the waste of replay under contention), and that this situation is rare. To the extent that target programs may have long-lived locks (e.g. due to I/O or blocking calls), we expect these results apply.

8. Conclusion

In this paper we presented Autolocker, an automatic lock insertion algorithm, along with a prototype implementation. Our initial results are promising: Autolocker effectively supports existing, realistic applications with low overhead. We plan to improve Autolocker in a number of ways. First, we wish to infer data sharing rather than forcing the programmer to annotate it. This would eliminate race conditions. We also plan to augment Autolocker with advanced language features, such as parametric polymorphism and regions, to reduce the number of programs rejected due to potential deadlocks. Finally, we will explore hybrid solutions that combine optimistic and pessimistic concurrency control to obtain the best possible performance.

Given the importance that parallelism will surely play in future systems, we believe that it is important to consider a wide variety of techniques for handling synchronization. Autolocker is an unexplored point in this space. Based on our evaluation, pessimistic atomic sections provide good performance and compatibility without restrictions on I/O, while requiring only a moderate level of annotation from the programmer. In the future, we believe that pessimistic atomic sections will play an important role in improving concurrent systems.

References

[1] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05*, pages 316–327. Feb 2005.

[3] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2004.

[4] M. Christiaens and K. de Bosschere. TRaDE: A topological approach to on-the-fly race detection in java programs. In *Proc. of the Java Virtual Machine Research and Technology Symposium*, April 2001.

[5] Robert Ennals. Software transactional memory should not be obstruction-free. `http://www.cambridge.intel-research.net/~rennals/faststm.html`.

[6] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *PLDI '00*, pages 219–232, 2000.

[7] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04*, pages 256–267, 2004.

[8] Cormac Flanagan and Stephen N. Freund. Type Inference Against Races. In *SAS'04*, pages 116–132, 2004.

[9] Cormac Flanagan and Stephen N. Freund. Automatic Synchronization Correction. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[10] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type Inference for Atomicity. In *TLDI '05*, pages 47–58, 2005.

[11] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03*, pages 338–349, 2003.

[12] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[13] Keir Fraser and Tim Harris. Concurrent programming without locks. `http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free`.

[14] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM Research Laboratory, 1975. Report RJ 1654.

[15] Tim Harris. Exceptions and side-effects in atomic blocks. In *Proceedings of the 2004 Workshop on Concurrency and Synchronization in Java programs*, pages 46–53, Jul 2004. Proceedings published as Memorial University of Newfoundland CS Technical Report 2004-01.

[16] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03*, pages 388–402. Oct 2003.

[17] Tim Harris and Keir Fraser. Revocable locks for non-blocking programming. In *PPoPP '05*. Jun 2005.

[18] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05*. Jun 2005.

[19] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03*, pages 92–101. Jul 2003.

[20] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*, pages 289–300. May 1993.

[21] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, 1983.

[22] D. B. Lomet. Process Structuring, Synchronization, and Recovery using Atomic actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 128–137, 1977.

[23] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*, pages 267–275, 1996.

[24] Kevin E. Moore, Mark D. Hill, and David A. Wood. Thread-level transactional memory. Technical report, University of Wisconsin, Mar 2005. CS-TR-2005-1524.

[25] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02*, pages 213–228, 2002.

[26] C. Praun and T. Gross. Object race detection. In *OOPSLA '01*, pages 70–82, 2001.

[27] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS '02*, pages 5–17. Oct 2002.

[28] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05*, pages 494–505. Jun 2005.

[29] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 2002.

[30] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP '01*. Jun 2001.

[31] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated Type-Based Analysis of Data Races and Atomicity. In *PPoPP '05*, pages 83–94, 2005.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[33] E. Schonberg. On-the-fly detection of access anomalies. In *PLDI '89*, pages 285–297, 1989.

[34] Y. Yu, T. L. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. Technical report, Microsoft Research, 2005. MSR-TR-2005-54.

$$
\begin{array}{cc}
\text{(LVALUE)} & \text{(ACQ)} \\[4pt]
\dfrac{\begin{array}{c}(\Sigma, \sigma, \kappa, q) \xrightarrow{lv} \ell \\ \sigma(\ell) = (v, \tau, L) \qquad L \subseteq \kappa\end{array}}{(\Sigma, \sigma, \kappa, q) \xrightarrow{e} v} &
\dfrac{\begin{array}{c}(\Sigma, \sigma, \kappa, q) \xrightarrow{lv} \ell \qquad \kappa' = \kappa \cup \{\ell\} \\ \ell \notin \kappa \Rightarrow \forall \ell' \in \kappa.\ \sigma(\ell') \prec \sigma(\ell)\end{array}}{(\Sigma, \sigma, \kappa, \mathrm{acq}\ q) \xrightarrow{s} (\sigma, \kappa', \mathrm{skip})}
\end{array}
$$

Figure 8. An excerpt of the operational semantics of LockC.

A. Soundness

In this section, we outline a proof that the LockC type system (Figure 3) ensures that well-typed and well-formed programs do not go wrong according to an operational semantics we have formulated. This proof assumes that a lock order $<$ has already been specified. It must be the same lock order as is used in the well-formedness checks.

Our operational semantics uses several mappings. Σ is an environment that maps variable names to the locations where they are stored in the heap. σ is a heap that maps locations to the values stored there. Technically, for any location ℓ, $\sigma(\ell) = (v, \tau, L)$, where v is a value of type τ. The location is protected by locks in set L. The set of locks that are currently held is stored in set κ.

We define the operational semantics via three relations. We use big-step rules for the evaluation of lvalues and expressions and small-step rules for statements.

$$
\begin{aligned}
(\Sigma, \sigma, \kappa, q) &\xrightarrow{lv} \langle \ell : \text{location} \rangle \\
(\Sigma, \sigma, \kappa, e) &\xrightarrow{e} \langle v : \text{value} \rangle \\
(\Sigma, \sigma, \kappa, s) &\xrightarrow{s} (\sigma', \kappa', s')
\end{aligned}
$$

The two most important rules in the semantics are shown in Figure 8. The rule for evaluating lvalues checks that all locks that protect a given memory location are contained in the set of currently held locks κ. The rule for acquiring locks adds the new lock to κ. It also checks that if the lock is not already held then no held lock should precede it in the lock order. This check is made using $\prec$, which simply finds the IDs of two locks based on their heap entries and then compares them using the standard lock order $<$.

Since LockC uses assume statements and non-determinism for control flow, there is the possibility that an ill-formed program will typecheck but will fail to evaluate. For example, the program "assume 0" cannot be evaluated even though it typechecks. To avoid this problem, we add an extra validity requirement beyond type checking to ensure that assume statements only occur in valid places. We call this requirement SyntaxValid(s). It checks that all repeat statements begin with an assume statement, and that choice statements begin with contradictory assume statements. No other assume statements are allowed.

We use a standard progress + preservation argument to prove soundness. As is typical in such proofs, we maintain a correspondence between the "runtime information" Σ, σ, and κ and the "static information" consisting of Γ and the computation history. First, we ensure that Σ and σ are always well-typed with respect to Γ. We also ensure that whatever lock protections appear in the types in Γ are reflected in the L components of the heap entries in σ.

However, the most important correspondence is between κ and the computation history. The history represents a static approximation of the set of locks that are held by the program. The (LVALUE) rule of the operational semantics requires that we have a static *under*-approximation of the set of held locks to guarantee that the program holds at least the ones that are needed. The (ACQ) rule forces us to have a static *over*-approximation of the held locks to ensure that the program holds no locks that are ordered before the one being acquired. We connect H to κ by ensuring that some string h in $L(H)$ (i.e., the history of some path in the program) approximates κ. Because it includes kill information, the string h is both an under-approximation and an over-approximation of κ.

The under-approximation is obtained from h by determining the set of locks that are acquired and then never killed. Stated formally,

$$
\kappa \supseteq \{\ell : \exists q.\ h \in L(-\overset{\oplus}{q}[\overset{\oslash}{q}]^*) \wedge (\Sigma, \sigma, \kappa, q) \xrightarrow{lv} \ell\}.
$$

That is, every lock lvalue that is acquired and not killed should evaluate to a location that is in κ.

The over-approximation is actually an over-approximation of lock IDs. We simply find the IDs of all lock lvalues that are acquired in h. Let $\mathrm{locids}(\sigma, \kappa)$ be the set of lock IDs of locks in κ. Then the formal statement of the over-approximation is

$$
\mathrm{locids}(\sigma, \kappa) \subseteq \{\mathrm{id}(\Gamma, q) : h \in L(-\overset{\oplus}{q}-)\}.
$$

Recall that $\mathrm{id}(\Gamma, q)$ is simply the ID of the lock lvalue q. The condition means that all the locks that are held in κ are represented by a lock acquired in h with the same ID.

We combine the heap validity condition with the under-approximation and over-approximation conditions on κ and H to form a new relation $\sim$. Formally,

$$
\begin{aligned}
(\Gamma, H) \sim (\Sigma, \sigma, \kappa) \equiv\ & \mathrm{HeapValid}(\Gamma, \Sigma, \sigma) \\
& \wedge \mathrm{UnderApprox}(H, \kappa) \\
& \wedge \mathrm{OverApprox}(H, \kappa)
\end{aligned}
$$

We use this relation to state the progress and preservation lemmas formally.

LEMMA 2 (Progress). *If* $\Gamma \vdash_s s : H$ *and SyntaxValid(s) and* $(\Gamma, H_0) \sim (\Sigma, \sigma, \kappa)$ *and* $WF(H_0 \cdot H, <, \Gamma)$ *then there exist* σ', κ', s' *such that* $(\Sigma, \sigma, \kappa, s) \xrightarrow{s} (\sigma', \kappa', s')$.

In this lemma, the history H_0 is the initial history that brought the program into a state with lock set κ. The history H summarizes any changes caused by the statement s. The entire history, $H_0 \cdot H$, must be well-formed for s to make progress with lock set κ.

LEMMA 3 (Preservation). *If* $\Gamma \vdash_s s : H$ *and SyntaxValid(s) and* $(\Gamma, H_0) \sim (\Sigma, \sigma, \kappa)$ *and* $WF(H_0 \cdot H, <, \Gamma)$ *and* $(\Sigma, \sigma, \kappa, s) \xrightarrow{s} (\sigma', \kappa', s')$ *then there exist* H_1, H_2 *such that* $\Gamma \vdash_s s' : H_2$ *and SyntaxValid(s') and* $L(H_1 \cdot H_2) \subseteq L(H)$ *and* $(\Gamma, H_0 \cdot H_1) \sim (\Sigma, \sigma', \kappa')$.

In this lemma, H_0 is again the initial history. The statement s proceeds to s' by the operational semantics. Since our rules for statements are small-step, s' is in some sense the remaining work to be completed for s. Thus, we divide the history H for s into two parts, H_1 and H_2. H_1 is the history completed during the step from s to s' and H_2 is the history that must be completed by evaluation of s'. That means that $H_0 \cdot H_1$ is the history for lock set κ', which was reached after the step to s'. And the new statement s' must type check with history H_2, since that is the work that remains to be done.

The proofs of progress and preservation are fairly typical, although they rely on the correctness of the killed function in the typing rule for assignments. We omit the details.