## 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. But existing proposals have several practical problems (high overhead, poor interaction with I/O). We present pessimistic atomic sections, a fresh approach that retains many advantages of standard atomic sections without sacrificing performance or compatibility. Pessimistic atomic sections employ the locking mechanisms familiar to programmers, while relieving them of most of the burdens of acquiring locks at the right time and avoiding deadlock. 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 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 prove the soundness of Autolocker’s core algorithm, and evaluate it on two high-performance web servers, with the larger having over 50,000 lines of C code. Autolocker incurs less than 5% overhead for the servers, and outperforms optimistic atomic sections on microbenchmarks in the presence of updates.

## 1. Introduction

Writing parallel and concurrent systems programs has historically been a difficult problem with few satisfactory solutions. Often a lot of effort goes into designing and enforcing complex synchronization policies and fighting bizarre bugs. Moreover every refactoring of the synchronization scheme for better performance involves tedious and error-prone code changes. We propose a new programming concept called a pessimistic atomic section to alleviate the problem. 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. The performance of a pessimistic atomic section can be tuned without affecting the correctness of the underlying code, and can be composed to create more modular programs. This paper evaluates pessimistic atomic sections via a prototype compiler called Autolocker that translates, using a provably sound to-source transformation that takes this code and adds necessary lock acquisitions and other bookkeeping operations provided by the Autolocker runtime library:

```c
void f() {
    struct { mutex rlk; int f; } rec;
    int v1;
    mutex glk;
    atomic {
        v1 = 3; 
        rec.f = v1;
    }
}
```

Like conventional locking, lock variables such as glk are written explicitly by the programmer. Unlike conventional locking, the programmer must also declare that this lock, a mutex, protects the global v1 variable. Similarly, the f field of any record of type struct rec is protected by the rlk field. Autolocker is a source-to-source transformation that takes this code and adds necessary lock acquisitions and other bookkeeping operations provided by the Autolocker runtime library:

```c
mutex glk;
int v1;
struct { mutex rlk; int f; } rec;
void f() {
    begin_atomic();
    acquire_lock(&glk);
    v1 = 3;
    acquire_lock(&rec.rlk);
    rec.f = v1;
    end_atomic();
}
```

Pessimistic atomic sections implement strict two-phase locking to provide database-level isolation; locks are acquired throughout a transaction and then released at the end. Atomic sections may be nested. Locks are dropped when the outermost section finishes. Additionally, Autolocker ensures that locks are acquired in a deadlock-free order.

### 1.1 Benefits

We are not the first to recognize the problems of manual locking. As an alternative, researchers have proposed optimistic implementations of atomic sections, relying either on transactional memory libraries [7, 9, 10, 11] or on experimental transactional hardware [2, 14, 18]. Unlike lock-based systems, which handle data contention via waiting, these techniques roll back and repeat an atomic section each time contention is detected. Because of rollback, every action performed by an optimistic atomic section must be reversible. This is a problem for systems that perform I/O inside atomic sections (such as one of the web servers we tested).

We believe that pessimistic and optimistic atomic sections are both potentially useful techniques for shared data access. 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 for systems with limited resources [1]. We believe that pessimistic atomic sections do have a number of practical advantages over optimistic ones, as well as over standard manual locking. Table 1 shows a breakdown of the differences between the three approaches to concurrency.

**Simplicity and correctness.** Atomic sections, both pessimistic and optimistic, provide clear usability benefits over manual locking. In an atomic section, the programmer is not responsible for
acquiring locks. Particularly in large systems with complex synchronization, this saves time and results in fewer bugs. Deadlocks are completely eliminated and race conditions are less likely, since these errors are caused by failing to acquire locks correctly.

**Practicality and compatibility.** Perhaps the most serious problem with 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 send 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 [8]. 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.

Existing code also causes problems for optimistic atomic sections. Nearly every existing system and library that takes advantage of concurrency already uses locks. Pessimistic atomic sections work well in such an environment because they can be introduced gradually. They require annotations about the existing locking policy, including lock protections and ordering, but they permit code using manual locking to be freely intermixed with code controlled by Autolocker.

Optimistic schemes also allow locking and atomic sections to be mixed, but in a more restricted way [9]. A memory location is either manually locked or it is controlled optimistically, but not both. Due to the way that most programs are structured, this effectively means that entire subsystems must be converted to optimistic control all at once. Consider the example of writing a device driver using atomic sections and integrating it into an existing operating system. Most of the driver’s data structures will be controlled by the operating system with locking. Optimistic schemes require the programmer to use manual locking when accessing this data. In contrast, Autolocker integrates seamlessly given the existing locking policy.

**Control.** Pessimistic atomic sections provide a useful separation between performance and correctness: performance can be tuned without introducing deadlocks or data races. The most important performance tradeoff in lock-based programming 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 causes fewer lock acquisitions (less overhead) but may lead to lock contention (extra waiting for locks). Fine-grained policies use more locks to protect data, so there is less likelihood that two accesses will acquire the same lock (less waiting). However, more locks are acquired overall, which can be inefficient.

When using Autolocker, the locking policy is determined entirely by lock protection annotations. By introducing more lock variables and changing data protections, the programmer can make the locking policy more or less fine-grained. Since these changes do not affect the code at all, or the sharing of data, they do not affect correctness of the program at all. Using manual locking, changing locking granularity is very painful. Each modification induces wide-ranging changes across the codebase, 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.

Optimistic atomic sections offer little control over performance aside from managing how conflicts are dealt with. A simple explanation of optimistic concurrency control is that it provides a maximally fine-grained sharing policy at the cost of very high overhead, at least for the software-based implementations. Unfortunately, this extreme trade-off is often suboptimal except when the level of concurrency is very high.

**Modularity.** Locking, like memory management and logging, is a cross-cutting concern. When programmers weave such concerns directly into their code, they make it less modular. Libraries that use one locking discipline are incompatible with code that uses another. The unrestricted combination of modules that use slightly different locking disciplines can lead to subtle bugs. The use of atomic sections eliminates these bugs, since synchronization happens automatically. The recent migration of many systems from manual memory management to garbage collection shows that automatic handling of cross-cutting concerns can yield huge benefits. We hope to realize similar gains with Autolocker.

### 1.2 Caveats

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. We found that annotations were not a burden at all when annotating performance-critical code. As we noted above, annotations are significantly easier to use than manually rewriting locking code to change the granularity.

However, most code is not performance-critical, and in this case annotations can be annoying. The current prototype requires that all shared variables be annotated with locks even when performance is not an issue. To handle these cases, we have designed an algorithm to infer lock protection annotations. Unfortunately, this algorithm still needs to know whether a variable is shared or not. For this information, we plan to use existing escape analyses (e.g., [19]) that determine whether data is accessible from multiple threads at once. Neither the protection inference nor the escape analysis has been implemented yet, but we believe they will eliminate the vast majority of non-performance related annotations.

Besides not needing annotations, using escape analysis to infer data sharing has the benefit that the information is guaranteed to be correct. Users can make mistakes when they write protection annotations, and these mistakes may lead to race conditions. Autolocker guarantees the absence of races only with respect to the annotations it is given. This qualification applies to any use of the term data race or race detection throughout the paper. It means that programs compiled using Autolocker never access a memory location without holding the lock protecting it, but they may race to access a memory location that is shared but not protected. Since escape analysis is conservative, it can guarantee that programs generated by Autolocker have no race conditions of any kind.

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 our largest evaluation, Autolocker rejected a module

<table>
<thead>
<tr>
<th>Approach</th>
<th>Programmer effort</th>
<th>Restrictions</th>
<th>Performance</th>
<th>Compatibility</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>acquire statements</td>
<td>none</td>
<td>total control</td>
<td>—</td>
<td>no deadlocks/no races</td>
</tr>
<tr>
<td>Optimistic</td>
<td>atomic blocks</td>
<td>poor I/O support</td>
<td>contention managers</td>
<td>can use locks in atomic sections</td>
<td>—</td>
</tr>
<tr>
<td>Pessimistic</td>
<td>atomic blocks &amp; lock protections</td>
<td>deadlock check may fail</td>
<td>granularity control</td>
<td>existing locks handled seamlessly</td>
<td>no deadlocks/no protection violations</td>
</tr>
</tbody>
</table>

Table 1. A comparison of manual locking, optimistic atomic sections, and pessimistic atomic sections.
only in 4 of 82 cases. 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 problem in Section 6, including a proposed improvement to the analysis that should accept the 4 previously rejected modules.

1.3 Contributions

There are two main contributions in this paper. In Sections 3 and 4, we present the algorithmic framework that underlies the Autolocker algorithm. We prove that the algorithm generates programs that are free of deadlocks and races (as defined above). The second contribution is the Autolocker tool itself, which is described in Section 5. Autolocker includes a number of extensions outside of the theoretical framework that are necessary to process real programs. In Section 6, we evaluate the success of Autolocker on several benchmarks, including two high-performance web servers, which reveal low overhead (below 5%), and good performance compared to software transactions (optimistic atomic sections). We discuss related work in Section 7 and conclude in Section 8.

2. Overview

We use a language called SimpleC (Figure 1) to formally describe the Autolocker tool. Like C, SimpleC programs are made up of statements containing expressions and lvalues. However, expressions and lvalues are side-effect-free. SimpleC does not permit pointer arithmetic or casting. Integers, pointers, and records are the only valid types. Our specification of SimpleC does not include variable declarations; it is assumed that the set of variables and their types is known. Functions are omitted for simplicity.

SimpleC provides statements to begin and end atomic sections (begin/end) and to acquire locks (acq). Locks are accumulated throughout the course of an atomic section and are not released until the end of the outermost one, as mandated by strict two-phase locking. Threads are not modeled explicitly and are not released until the end of the outermost one, as mandated by strict two-phase locking. Autolocker conceives SimpleC into programs using several analyses that we will describe throughout the paper.

Types in SimpleC include lock protection annotations in brackets. Every memory location has an optional lock to protect it; 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. The simplest lock name is \( \bot \), which means that no protection is necessary. 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 \( \{ m : \text{lock}[\text{mylock}], f : \text{int}[m] \} \), then \( x.f \) is protected by \( x.m \). Lock annotations like this are very important, since using only global locks would force programmers to use very coarse-grained locking policies.

SimpleC requires that locks acquisitions be ordered within every atomic section. This order is arbitrary, but it must be acyclic in order to prevent deadlocks. This order is expressed in terms of lock labels provided with each lock type. For instance, in the example above, \( x.m \)'s label was “mylock”. Labels can be inferred using alias analysis, although having meaningful lock labels can be useful for generating informative error messages. In the rest of this paper, we assume lock labels are inferred and omit them from source code examples.

Section 3 presents a formal semantics for SimpleC, which gives precise definitions of race conditions and deadlocks. The operational semantics for SimpleC ensures that programs that do not follow the lock order, or access unprotected data will “go wrong.” The system for SimpleC ensures that type-correct programs never go wrong. In Section 4, we present our Autolocker algorithm which places acquire operations and generates a lock order, and proves that it generates type-correct programs (it will however reject some programs). Thus, the output of Autolocker is free of deadlocks and race conditions.

3. SimpleC Language

A program written in SimpleC consists of a typing context containing all the variables and their types, a statement to execute, and a lock ordering. The lock ordering is a total order on lock labels that describes the order in which locks are acquired in atomic sections. Just as it is illegal to use a pointer variable as if it were an integer, it is illegal to acquire locks out of order. The next two sections describe operational and static semantics for SimpleC.

3.1 Operational Semantics

Besides simple type safety, the operational semantics for SimpleC ensure that programs are free of deadlocks and race conditions. In this section, we describe the exact definition of a deadlock and a race condition, the invariants that are needed in order to check for errors, as well as the individual evaluation rules for SimpleC constructs. Figure 2 shows the main rules of the operational semantics.

Values. SimpleC values are tagged with their type. A value may be an integer, a pointer to a memory location, or a record containing fields. A memory location is simply an integer. A record is a mapping from fields to memory locations storing the field values. Using ML notation:

\[
\begin{align*}
|v| &= \text{Int}(\alpha) \mid \text{Ptr}(\ell) \mid \text{Rec}(F) \\
F &= \text{FieldName} \rightarrow \ell
\end{align*}
\]

Heap. The heap maps memory location to tripled containing a value, the type of the value, and a set of locks protecting the location. The malloc statement (rule not shown) is responsible for the layout of the heap. It allocates heap locations for a new value of a given type. The rule is fairly large, since it must interpret lock names in an environment that includes both global locks as well as locks stored in record fields. When a record is allocated, the lock fields in the record are allocated in the heap and then added to the environment, which is used to finish allocating the remaining record fields.

Race conditions. We have already mentioned the caveat that failing to mark a shared location with a lock may lead to a race condition; since data sharing is not modeled in the semantics at all (there...
are no threads) this type of error will pass unnoticed. We rely either on the user or on a conservative escape analysis to protect data correctly. We do however ensure that all protecting locks are held when memory is accessed.

Checking these protections necessitates knowing the set of locks held at all times. The set of held locks is stored in a lockset, written $\kappa$, which flows through each transition rule. The set of locks protecting each memory location is stored in the heap. The type system tracks locks by the lvalue used to access the location based on its type. The lvalue expression can be evaluated only if all those locks are held in the set $\kappa$. The (ASSN) rule ensures that the locks protecting the location being assigned to are held as well.

Locks are added to $\kappa$ via the (ACQ) rule. Information about the current nesting level of atomic sections flows through the transition rules for statements. All locks are dropped when the outermost atomic section ends.

Deadlocks. The semantics ensures the absence of deadlocks by enforcing an order in which locks must be acquired. This condition, of course, does not preclude the program from looping infinitely or otherwise holding a lock forever. However, it does prevent many common programming errors involving locks (particularly those that are introduced during performance tuning).

The lock order is a total order on lock labels. For two labels where $\text{label}_1 < \text{label}_2$, the semantics requires that locks labeled with $\text{label}_1$ cannot be acquired after locks labeled with $\text{label}_2$. This property is enforced in the (ACQ) rule. The acquire statement demands that, unless the new lock is already held, it must be ordered after every currently held lock. The Precedes predicate uses type information stored in the heap to look up lock labels.

3.2 Type System

The type system for SimpleC (Figure 3) ensures that well-typed programs never suffer from race conditions or deadlocks. Like a standard type system, each rule ensures that the values used in a construct have the right types. In addition, the typing rules generate a regular expression, called a history, that summarizes the concurrency-related actions that the program performs. A program is type-correct if it has a type derivation with history $H$, and $H$ is well-formed. The well-formedness check ensures that the program’s actions do not lead to a data race or a deadlock.

Locks. The type system tracks locks by the lvalue used to access them, rather than by the lock labels used by the operational semantics — these labels can map to many locks and are therefore not precise enough for type checking purposes.

The typing rules for expressions and lvalues determine the set $\kappa$ of lock labels that must be held for valid execution. Determining the lvalue of a global lock is straightforward. However, when the lock is embedded inside a structure, the process becomes more complex. Consider the following example:

\[
\begin{align*}
x & : \{ L : \text{lock} \} \\
y & = x.v + 1
\end{align*}
\]
Figure 3. An excerpt of the type system of the language of SimpleC.
The lvalue of the lock needed to access \( x.v \) is \( x.L \). The typing rules for lvalues and expressions include a lock naming environment to help resolve such accesses. This environment maps lock names (such as \( L \)) to their respective lvalues \((x.v)\). The intenv function creates an initial context that maps global lock names to lvalues. The \((\text{FIELD})\) rule uses the addenv function to add locks found in records to the context. Wherever types are used (such as during assignment or when determining the locks protecting a type), they are first normalized by looking up all lock names in the environment.

Assignments may cause a particular lock lvalue to become in-accessible. This is enforced by the well-formedness condition on histories (see below).

**Histories.** The computation history of a statement is a regular expression that summarizes the effect of the statement on the program’s locks. Each string in the language of the regular expression describes a path through the statement. Histories add a measure of path sensitivity to the type system and they concisely describe a potentially infinite number of paths in the program. More importantly, though, they separate the task of proving that a program respects the types of values (traditional type checking) from the task of checking for race conditions and deadlocks. In the next section, we show that the Autolocker algorithm accepts a SimpleC_{inf} program as input that already has a valid type derivation and it converts it to a SimpleC program that also has a well-formed history. Histories are also convenient for describing the Autolocker algorithm.

In full generality, the type system should generate one history for each atomic section. For a program to be correct, all of the histories that it generates should be well-formed. However, to simplify the presentation of the type system, we assume that a program consists of only one atomic section; thus, we have omitted the rules for begin and end statements. The full type system supporting multiple atomic sections is more verbose but not inherently more complex.

**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 ignores top-level lock protections (since they are not changed by the assignment) but it requires that protections inside pointers match exactly, as is standard in subtyping for mutable references.

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. For example, a program that acquires a lock with lvalue \( q \), makes an assignment that changes the value of \( q \) and then accesses a memory location that is protected by \( q \) is not safe. The program no longer holds the concrete lock that \( q \) evaluates to, which is what the operational semantics requires.

To preserve safety, the \((\text{ASSN})\) rule records in the history all the lvalues \( q \) whose value might be affected by the assignment, as defined by the killed function. We do not define the killed function. It may be implemented by any sound alias analysis. Our soundness proof simply requires that any lvalue that is not killed is unaffected by the assignment statement. Well-formedness rules WF2 and WF3 (see below) ensure that killed lvalues are not misused.

**Notation and Example** Histories are regular expressions over the following alphabet, where \( q \) is any lvalue:

- \( \oplus \) Lock with lvalue \( q \) is acquired.
- \( \ominus \) Value protected by \( q \) is used by a statement.
- \( \ominus \) Lock with lvalue \( q \) is killed by an assignment.

For notational convenience, we use a dash (\(-\)) instead of \( \ominus \) to represent an arbitrarily long string of symbols. For an example history, consider the following program.

\[
x : \{ L : \text{lock}, v : \text{int}[L] \} ~ \text{ref} \perp
\]

\[
\text{acq} \ (\ast x).L;
\]

\[
y = (\ast x).v + 1;
\]

if \( e \) then \( x = z \) else skip

Let \( q \) be the lvalue \((\ast x).L \). This program generates the path history \( \oplus q \ominus q \ominus q \).

**Well-formedness.** There are four well-formedness conditions. Two are related to data races and two to deadlocks. Since computation histories are written in terms of lock lvalues, and reasoning about deadlocks is done using lock labels, we use some additional notation to map between them.

\[
\text{label}(\Gamma, q) : L \Leftrightarrow \exists S, E. \Gamma \vdash_\circ q : \text{lock}[L]; S; E
\]

This function is well-defined since the typing rules admit only one type for any lvalue. Based on this function, we define some supplementary ordering relations. These relations are based on the same ordering \( < \) that was used in the operational semantics.

\[
q_1 <_r q_2 \equiv \text{label}(\Gamma, q_1) < \text{label}(\Gamma, q_2)
\]

\[
q_1 \leq_r q_2 \equiv q_1 <_r q_2 \lor \text{label}(\Gamma, q_1) = \text{label}(\Gamma, q_2)
\]

Using this notation, the four well-formedness conditions can be summarized as follows.

**WF1** — No memory access is unprotected. The first condition is that there is no history \( h \in L((\oplus q \ominus q \ominus q) \ominus q) \). A history of this form accesses a memory location that requires a lock \( q \) even though it has never acquired \( q \).

**WF2** — Protecting locks are never killed after acquisition. The second condition is that \( h \notin L(\ominus q \ominus q \ominus q) \ominus q \ominus q \ominus q \). If a lock is killed, then the previous acquisition of the lock is not helpful for future memory accesses, since there is no guarantee that the acquisition and the access refer to the same lock.

**WF3** — Killed locks are never re-acquired. A lock that has been killed should never be reacquired. After the kill, the lock lvalue might evaluate to a different concrete lock than originally. These two concrete locks cannot be ordered with respect to each other (since they have the same lock label), and the second acquisition could deadlock. Thus, histories matching \((\ominus q \ominus q \ominus q) \ominus q \ominus q \ominus q \) are forbidden.

**WF4** — Locks are acquired in order. For a history \( h \in L((\ominus q \ominus q \ominus q) \ominus q \ominus q \ominus q) \ominus q \ominus q \), lock \( q_2 \) is definitely acquired before \( q_1 \) (since no acquisition of \( q_1 \) happens first). If \( q_1 <_r q_2 \) in the locking order, then there is a problem. Thus, these histories are excluded.

### 3.3 Soundness

The correctness proof for Autolocker relies on the soundness of the SimpleC type system. The soundness proof is slightly unusual because of the well-formedness conditions. We define a relation \( \sim \) that relates information from the type system to information from the operational semantics. The most obvious condition is that the heap must be well-typed according to the typing context. The predicate \( \text{HeapValid}(\Gamma, S, \sigma) \) checks this property. Its definition is fairly typical, although it also checks that locks protections are correctly placed in the heap.

Additionally, the \( \sim \) relation must ensure that the computation history matches the set of locks that are actually held by the program. Due to the way that the type system simultaneously checks for race conditions and deadlocks, computation histories must be able to represent both an underapproximation and an overapproximation of the current lockset. We obtain the underapproximation by aggregating the lvalues of all locks in the history that have been acquired and not killed. We obtain an overapproximation of the set of labels of held locks by considering lock acquires only. To de-
termine the labels of locks in the current lockset, we use a simple function, defined as:

\[
\text{labels}(\sigma, \kappa) = \{ L : \exists \ell \in \kappa, v. \sigma(\ell) = (v, \text{lock}[L])\}
\]

Formally, \(\sim\) is defined as follows:

\[
(\Gamma, H, n) \sim (\Sigma, \sigma, \kappa) \equiv \\text{HeapValid}(\Gamma, \Sigma, \sigma) \\
\wedge n > 0 \implies \exists h \in L(H). \\
\kappa \supseteq \{ \ell : \exists \mathcal{h}. h \in L(\neg (\sigma_0(h) \cap \sigma_1(h)) \wedge (\Sigma, \sigma, \kappa, q) \Downarrow \ell) \} \\
\wedge \text{labels}(\sigma, \kappa) \subseteq \{ \text{label}(\Gamma, q) : q \in h \}
\]

The soundness proof itself is in a typical progress + preservation style. The proof itself is omitted.

**Lemma 1 (Progress).** If \(\Gamma; n \vdash_s s : H; n’\) and \((\Gamma, H_0, n) \sim (\Sigma, \sigma, \kappa)\) and \(WF(H_0; H, <, \Gamma)\), then there exist \(\sigma’, \kappa’, s’\) such that \((\Sigma, <, \sigma, \kappa, n, s) \Rightarrow (\sigma’, \kappa’, n’, s’)\).

**Lemma 2 (Preservation).** If \(\Gamma; n \vdash_s s : H; n’\) and \((\Gamma, H_0, n) \sim (\Sigma, \sigma, \kappa)\) and \(WF(H_0; H, <, \Gamma)\), then there exist \(H_1, H_2\) such that \(\Gamma; n’ \vdash s’ : H_2; n’’\) and \(H_1 \cdot H_2 \subseteq H\) and \((\Gamma, H_0; H_1, n’’) \sim (\Sigma, \sigma’, \kappa’).

4. Acquisition Placement

The main purpose of the Autolocker tool 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. It does so by inferring the program’s data dependencies, which cause dependencies between locks. These dependencies form a partial order on the lock labels. If this order is cyclic, then the program is rejected. Otherwise, the partial order is converted to a total order using a topological sort. Locks are acquired in this order.

2. In the second stage, lock acquisitions are placed throughout the program. 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 modified SimpleC program that contains lock acquisitions.

In the rest of the section, we describe the two phases of the algorithm in greater detail. Then we prove that the placement algorithm always generates type-correct programs. Thus, it never creates race conditions or deadlocks.

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:

\[
\begin{align*}
T &= \{ L : \text{lock}, v : \text{init}[L] \} \\
G : \text{lock}; x : T &\text{ ref}[G]; y : T \text{ ref}[] \\
y &= x; \\
(*y).v &= 1
\end{align*}
\]

The first statement in this example requires that the lock \(G\) be held. The second statement requires that \((*y).L\) be held. It is impossible to acquire \((*y).L\) before \(G\), because the value of \(y\) depends on the value of \(x\), and \(x\) cannot be read unless \(G\) is held.

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. These data dependencies are caused by 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. There are two classes of data dependencies. The first class consists of purely local dependencies, where accessing one lock lvalue requires that another lock be held. To describe these self-dependencies, we introduce the sets \(\text{use}(q)\) and \(\text{use}(s)\) that contain the lock lvalues that must be held in order for lvalues or statements to execute. These sets can be computed using the type checking rules for expressions and lvalues (Figure 3). The set of self-dependencies for a statement is defined as:

\[
\text{selfdeps}(s) = \{(q_1, q_2) : q_1 \in \text{use}(q_2), q_2 \in \text{use}(s)\}
\]

It contains a dependency edge from \(q_1\) to \(q_2\) if \(q_2\) is needed by \(s\) and \(q_1\) is needed by \(q_2\).

Besides self-dependencies, the lock order inference must account for dependencies created by assignment statements. These dependencies are embedded inside the computation histories that are determined by the type checking rules. Imagine a computation history \(h \in L(-q_1 - q_2 - q_2-\)\). In this history, lock \(q_1\) must be acquired before \(q_2\), since \(q_2\) is killed after \(q_1\) must already have been acquired. Even if \(q_2\) were acquired before \(q_1\), the kill means that the lock required later on might potentially be different than the one that was acquired.

More formally, we define the set of dependencies due to assignments using an intersection of regular languages. For a statement \(s\), let \(H(s)\) be the computation history generated by \(s\). This history is computed using the type checking rules for statements. Then:

\[
\text{killdeps}(s) = \{ (q_1, q_2) : H(s) \cap L(-q_1 - q_2 - q_2-) \neq \emptyset \}
\]

The set of all dependencies for a statement is simply \(\text{selfdeps}(s) \cup \text{killdeps}(s)\). These dependency edges form a directed graph. In the case when it is acyclic, we use topological sort to generate a total order on lock labels. (It is easy to convert the graph nodes from lvalues to labels.) If the graph is cyclic, the program is rejected.

4.2 Acquisition Placement

The previous algorithm generates a total order on lock labels. Next, the lock placement algorithm must acquire locks in the given order. We use computation histories in order to place the locks. For simplicity, we will assume that each statement includes a distinct statement number to identify it. We augment the computation histories in order to place the locks. For simplicity, we will assume that each statement includes a distinct statement number to identify it. We augment the computation histories in order to place the locks. For simplicity, we will assume that each statement includes a distinct statement number to identify it.

For each statement \(i : s\) that requires a lock \(q\), we insert the statement \(acq[q]\) just before \(i\). This algorithm ensures that no memory location is accessed without the proper lock. However, it may order locks improperly. To solve the problem, we also add acquisitions of \(q\) before any previously inserted acquire statements for locks \(q’\), where \(q < r q’\). This ensures that locks are always acquired in order. The acquisitions that are inserted due to a particular statement are computed as \(\text{ins}(s, i)\), where \(I\) is the set of insertions for previous statements. The result of \(\text{ins}\) is a set of statement number and lock lvalue pairs. It is defined as follows:

\[
\text{ins}(i : s, I) = I \cup \bigcup_{q \in \text{use}(s)} \{ ((i, q)) \cup ((j, q) : (j, q’) \in I, q < r q’) \}
\]
Just like the use function, we overload \( \text{ins} \) to handle expressions as well as statements. The \( \text{ins} \) function works only when there is no control flow. For entire programs, we define \( \mathcal{T}(s, I) \) as follows:

\[
\mathcal{T}(i : e \rightarrow e, I) = \text{ins}(i : e, I)
\]

\[
\mathcal{T}(s_1; s_2, I) = \mathcal{T}(s_2, \mathcal{T}(s_1, I))
\]

\[
\mathcal{T}(i : \text{while } e \text{ do } s, I) = \mathcal{T}(s, \text{ins}(i : e, I))
\]

\[
\mathcal{T}(i : \text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow I) = \mathcal{T}(s_1, \text{ins}(i : e, I)) \cup \mathcal{T}(s_2, \text{ins}(i : e, I))
\]

Based on the set of insertion locations, we insert lock acquisitions for statement \( i \) just before \( i \). If there are multiple locks to be acquired for a single statement, they are sorted according to the lock ordering. The transformation to insert lock acquisitions at sites \( I \) is computed by \( \mathcal{T}(s, I) \) as just described. We omit the formal definition.

Assuming that a program \( s \) has a valid typing derivation, the transformation \( \mathcal{T} \) generates a program that is not only well-typed, but also has a well-formed history, as we prove below. Based on our previous soundness theorem, these two conditions guarantee that the transformed program is free of deadlocks and race conditions. Before proving well-formedness, we need a lemma that describes the way that histories behave under transformation.

**Lemma 3 (Histories).** Let \( H \) be the computation history of a program \( s \), and let \( H' \) be the history of the transformed program \( \mathcal{T}(s, s(I, \emptyset)) \). Then the following properties hold.

(i) If there is a history \( h' \in L(H') \) such that \( h' \in L(-q_1 \sqsupseteq q_2 \sqsupseteq q_3 \cdots \sqsupseteq q_n) \), then there is a history \( h \in L(H) \) such that \( h \in L(-q_1 \sqsupseteq q_2 \sqsupseteq q_3 \cdots \sqsupseteq q_n) \). That is, if there is a dependency in a history in \( H' \), then the same dependency exists in \( H \).

(ii) If a history \( h' = h_0 q_1 h_1 \in L(H') \), then \( h_1 \) has the form \( h_1 = q_2 \cdots q_n h_2 \), where \( n \geq 0 \) and \( q_1 \sqsubseteq \Gamma \). In other words, every lock acquisition is always followed by more acquisitions and then a lock use, and all these locks are in order.

(iii) If there is a history \( h' = h_0 q_1 h_1 \in L(H') \), then there is also a history \( h'' = h_0 q_1 h_2 q_3 h_3 \in H' \). 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.

We use this lemma to prove that the history of the transformed program \( \mathcal{T}(s, s(I, \emptyset)) \) is well-formed, and thus that the transformed program is free of deadlocks and race conditions. Well-formedness consists of four properties shown in Figure 3. We sketch a proof of these four properties here.

**WF1** — No memory access is unprotected. Since the lock insertion algorithm always includes an element \((i, q)\) for every element \( i : q \) of the original history, we are guaranteed that every original memory access will be immediately preceded by the acquisition of the lock protecting the accessed location. Lock acquisitions may add additional lock uses. However, self-dependencies ensure that the acquisitions for these uses occur correctly.

**WF2** — Protecting locks are never killed after acquisition. Since an access to a lock \( q \) will always be preceded by an acquisition of \( q \), this condition is subsumed by a later one, that killed locks are never re-acquired.

**WF3** — Killed locks are never re-acquired. Assume that this happens. In this case, the transformed history contains a string \( h \in L(-q_1 \sqsupseteq q_2 \sqsupseteq q_3 \cdots) \). By part (ii) of the lemma, the initial acquisition must be followed by a use, so \( h \in L(-q_1 \sqsupseteq q_2 \sqsupseteq q_3 \cdots) \), where \( q_1 \sqsubseteq \Gamma q_2 \). Then by part (iii) of the lemma applied to the final lock acquisition, there is some other \( h' \in L(H') \) where \( h' \in L(-q_1 \sqsupseteq q_2 \sqsupseteq q_3 \cdots) \). This history contains a dependency \( q_2 \sqsubseteq \Gamma q_1 \), since \( q_2 \) is followed by a kill and then a use of \( q_1 \). We assumed that \( q_1 \sqsubseteq \Gamma q_2 \), so this is a contradiction. Thus, killed locks are never re-acquired.

**WF4** — Locks are acquired in order. When the lock insertion algorithm inserts an acquire statement, it also inserts acquires before every earlier acquisition of a lock that is greater in the lock ordering.

### 5. Extensions

Real programs are built using a variety of features that are inexpressible in SimpleC. 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. Condition variables have 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. A thread loops, waiting on a condition variable, until the predicate is satisfied. Any thread that updates a variable that might affect the predicate must also 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. [9]

#### 5.2 Prelocking

SimpleC 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 a predicate, the programmer may assign a local variable to point to one hash table or the other. Autolocker includes a special protection annotation written `$\text{ locked}$`. A variable marked `$\text{ locked}$` (a `prelocked variable`) may contain data that is protected by any arbitrary lock (according to the concrete semantics) as long as the lock is held throughout the lifetime of the variable.

Autolocker includes a subsumption rule so that data protected by a lock can be converted to `$\text{ locked}$` 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 `$\text{ locked}$` variable. In particular, global variables cannot be prelocked.
of locking. For libraries, this property adds a valuable form of modularity.

5.3 Reader/Writer Locks

Reader/writer locks are an extension to simple mutexes which allow 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 acquirees of that lock in that atomic section will be write-acquirees. Otherwise, the acquirees will be read-acquirees (acquiring a lock first for read, and then for write can lead to deadlock).

There is a complication in this algorithm when a function is called from two atomic sections. Even if the function only requires read access to a memory location, other code inside the atomic sections may access the location in different ways. To ensure that all lock acquisitions in a section are either for reading or for writing, each section begins by notifying the runtime about how reader/writer locks with a particular label should be acquired throughout the section. This ensures that all lock acquisitions for a particular lock are of the same type, thus preventing the possibility of deadlock.

5.4 Lock Subordination

In many cases, reader/writer locks are used in a hierarchical fashion. For example, a hash table may have a global lock protecting the hash table’s arrays, and individual locks protecting each bucket; furthermore, a bucket lock must not be acquired if the global lock is not held. As a consequence, if the program has acquired the global lock for write (e.g., because it’s creating a new bucket, or rehashing the table), it does not need to acquire the bucket locks. This organization increases the concurrency of operations that only modify individual buckets.

In Autolocker, a lock \( L \) can be declared as subordinated to another lock \( L’ \). When this is done, Autolocker will automatically acquire \( L \) whenever it would acquire \( L’ \); additionally, if \( L’ \) is known to be acquired for writes, the acquirees of \( L \) will be suppressed. Note that this also implies that \( L’ \) must come before \( L \) in the global lock order. Our current implementation of subordination 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 2500-line extension to the CIL C analysis framework [15]. Autolocker is run on a C program with atomic statements and lock declarations and outputs a C program with calls to the Autolocker runtime library.

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. The full implementation is in five stages:

1. The C program’s files are assembled into a single module using CIL’s file-merging capabilities. This is necessary to ensure a consistent program-wide lock order.
2. The set of locks that may be acquired or killed from each point until the end of the outermost atomic section is computed in a flow-sensitive, context-insensitive way.

3. An order for the program’s locks is computed. If such an order cannot be found, we reject the program as it may deadlock (Section 4.1).
4. Lock acquirees are added to the program (Section 4.2).
5. The set of locks definitely acquired at each point is computed in a flow-sensitive, intraprocedural way. Redundant acquirees are removed.

For call-graph construction, functions are grouped into equivalence classes as in a standard unification-based alias analysis. We rely on the user to group functions via checked annotations. Since few function pointers are called from atomic sections, not many annotations are necessary. In the case of external library functions, we unsoundly assume that they 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.

The call-graph is used to determine the set of locks that may be acquired or killed in an atomic section (essentially, the computation history). However, due to scoping issues with functions, local locks are killed at the end of a function. To callers, they are visible only by their lock labels. In some cases, this summarization may lead the lock ordering algorithm to be over conservative. In the rare case that it occurred, inlining solved the problem.

Every Autolocker program links to the libautolocker library, a simple runtime (900+ lines of C) that manages atomic sections and provides other utility functions. It uses a per-thread data structure to keep track of all locks currently held by a thread and other information like atomic-section nesting depth. Common techniques like pooling (for lock entries) are used to reduce overhead. It also has its own implementation of read/write locks, which uses spinlocks instead of mutexes to protect the read/write lock struct and a fast Linux kernel mechanism called futex [4] to communicate with other threads. This implementation proves to be faster in our experiment than the rw-lock in NPTL, the current pthread library on Linux. But as we did not spend a lot of time optimizing the library, there should be much room for further improvement.

We evaluated Autolocker using one microbenchmark, a highly concurrent hashtable, and two larger programs, the largest of which is over 50,000 lines of code. 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 hash table using Autolocker and compared it with three other versions, one using a global lock, one with finer-grained locking, and another using word-based software transactional memory [9]. The code is derived from the hash table in NSPR, a portable runtime used in the Mozilla Web browser.

The original code does not do any synchronization. The Autolocker-based version supports concurrent lookup and modification by using a two-level locking scheme. The whole hashtable is protected by a read/write lock and each hash bucket by a mutex. Lookups, insertions and removals first acquire the read lock on the whole hash table, then acquire the mutex on the specific bucket involved. Global operations like hash table growing/shrinking grab the write lock and operate on all buckets without locking the mutexes.

Implementing this scheme using Autolocker is relatively easy. First a read/write lock is added to the hash table struct, a mutex added to the bucket struct and certain fields declared as protected by these locks. Then the outermost operations like insertions/lookups are enclosed in an atomic section. The mutex is annotated as subordinated to the read/write lock (Section 5.4). One issue is that we had to separate out the code to actually grow and shrink the hash table as a separate atomic section
The fine-grained manual-locking version (Fine/Manual) uses the same locking scheme, and resolves the same issues in essentially the same way. In addition, we had to make sure every code path locks and unlocks properly, in particular where functions return prematurely due to error conditions. For two functions, this involves changing the functions’ interfaces to return the locked buckets because the bucket lock is obtained with these functions and must be unlocked in the outermost function to ensure atomicity.

The third version, coarse-grained manual-locking (Coarse/Manual) is very simple. The whole hash table is protected by a single mutex and every external operation locks the mutex.

The software transactional memory version (STM) replaces data structure reads and writes with calls to the STM library, and retries operations when commits fail. Because of limitations on the number of writes per transaction, this version does not support growing or shrinking the hash table (our benchmarks do not exercise these code paths).

Both Autolocker and STM had issues with freeing memory during transactions: Autolocker needs to access the locks acquired during the transaction at its ends; the STM system would write to freed objects at commit time. In both cases, the solution is to defer freeing operations to commit time.

The hash table benchmark is run on a 2-way Xeon 2.8Mhz (1 MB L2 cache), with Hyperthreading enabled, i.e. each CPU has two SMT threads. The OS is Fedora Core 3 Linux with kernel 2.6.10. In each experiment, the hash table is first populated with 100k elements with integer keys and values. We then measure the number of operations performed per second with different number of concurrent threads, under pure lookup and mixed update/lookup scenarios.

Figure 4 shows the results for lookups. The coarse-grained version (Coarse/Manual) has the highest throughput with one thread and degrades sharply when running with multiple threads, due to coarse sharing. The Autolocker version (Fine/Autolocker) and manual fine-grained version (Fine/Manual) both get modest speedup going from one to two threads but degrade thereafter. The performance gap between Fine/Autolocker and Fine/Manual is due to the extra overhead Autolocker introduces in keeping the atomic section and lock set data structures. The gap gets smaller as the number of threads increase (and drops below 5% for the real applications below). One reason for this is that most of Autolocker book keeping work is thread-local and parallel. For this read-only test, the STM version outperforms for multiple threads.

Figure 5 shows the results with 60% lookups and 40% insertion/removals. Both Fine/Autolocker and Fine/Manual have significantly higher performance than Coarse/Manual with multiple threads, while STM has lower throughput.

There is a large literature on constructing concurrent data structures including hash tables (e.g., [13, 22]). Our implementation is quite limited, and gains no significant speedup with multiple CPUs. This is partly because our hash table operations are very simple, making lock contention high. However this microbenchmark confirms that using Autolocker to implement non-trivial lock-based concurrent control is easier than manual locking. Moreover the overhead compared to the manual version is reasonable and reduces as parallelism increases.

6.2 Knot Web Server

Knot is an experimental threaded web server used in some of our previous work [23]. It contains 2500 lines of C code. Its use of locking is relatively simple. All global data, including a hash table for cached content and other statistics, are protected by a single mutex. Cache entries are immutable and thus need no locking. However, a reference count, protected by a per-entry mutex, is used to ensure proper freeing of entries. Rewriting knot to use Autolocker was simple. The only significant change was to manually inline a helper function in two places to resolve the ordering of two locks. In total, 9 variables/fields are declared as protected by mutexes, and 15 atomic sections are added.

We used the ab2 tool from Apache 2 to benchmark performance of the two versions of Knot in a Gigabit Ethernet LAN. The server is a dual-Xeon 2.4Mhz running Debian Linux 3 with kernel 2.6.12.3. Four other machines of the same model are used to generate load. To stress the synchronization part in Knot, we minimize the impact of disk and network I/O by serving a single 2-byte file. The throughput result is shown in Figure 6. As can be seen, The Autolocker version yields at most a 3.5% lower throughput. In a real application environment, higher network and disk I/O will make this gap smaller.
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 broadened somewhat, since Autolocker forces all atomic sections to begin and end in the same function. We broadened less 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, we printed a warning. This technique does not guarantee that all lock protection annotations are correct, but it does ensure that the modified program has at most as many race conditions as the original one did.\(^2\)

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 optimistic approaches).

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 particularly 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 other ways.

The remaining 23 of 175 trusted casts resulted 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. To compensate, we coarsened the locking granularity so that the analysis would succeed. Each of the four modules originally created networks of dynamically allocated objects. Each object in a network was 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. 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 that set of programs Autolocker accepts.

Performance results for AOLserver are shown in Figure 7. The experimental setup is the same as for Knot. The throughput penalty of Autolocker ranges from 2.0% to 4.9%. As with Knot, we expect real world performance of the two versions to be even closer.

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 [16, 20], happens-before relationships [3, 21], or both [24]. 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, Freund, Qadeer have explored atomicity for the purpose of finding race conditions in programs that use manual locking [6, 5]. They use static and dynamic analysis to ensure that the locking policy chosen by the programmer actually enforces atomicity constraints that have been added to the code. Their static checker is similar to Autolocker. Like our tool, it requires lock protection annotations from the programmer. But instead of checking for the atomicity property, Autolocker inserts lock acquisitions to enforce it. We have already described the ease-of-use and modularity benefits of automatic lock insertion.

Herlihy and Moss [12] proposed the first transactional memory hardware based on extending cache-coherency mechanisms; Ra-

\[^2\] Interestingly, although Autolocker is not designed to detect race conditions, we found several possible 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.
jwar and Goodman [17] used similar cache-coherency-based hard-
ware transaction support to optimistically execute lock-based pro-
grams. 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 lan-
guage. These limitations have been addressed in the more recent
work by Ananian et al [2], Moore et al [14] and Rajwar et al [18].
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 imple-
mented by Harris and Fraser [9, 7, 10] and Herlihy et al [11], run on
existing hardware. However, they have fairly high overhead. Fur-
thermore, both software and hardware transactional memory are
based on an optimistic concurrency model with rollback and thus
have problems with non-reversible actions such as I/O.

The relative merits of optimistic and pessimistic concurrency
control were the subject of many studies in the database litera-
ture [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 inser-
tion algorithm along with a prototype implementation. Our initial
results are promising: Autolocker effectively supports existing, re-
alistic applications with low overhead. We plan to improve Au-
tolocker in a number of ways. First, we wish to infer data sharing
rather than forcing the programmer to annotate it. This would eli-
minate all 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 un-
explored point in this space. Based on our evaluation, pessimistic
atomic sections provide good performance and compatibility with-
out restrictions on I/O, while requiring only a moderate level of
annotation from the programmer. In the future, we believe that pes-
simistic atomic sections will play an important role in improving
concurrent systems.

References

[1] Rakesh Agrawal, Michael I. Carey, and Miron Livny. Concurrency
control performance modeling: Alternatives and implications. ACM

Leiserson, and Sean Lie. Unbounded transactional memory.
In Proceedings of the 11th International Symposium on High-

to on-the-fly race detection in java programs. In Proc. of the Java

2004.

atomicity checker for multithreaded programs. In POPL ’04:
Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on
Principles of programming languages, pages 256–267, New York,

[6] Cormac Flanagan and Shaz Qadeer. A type and effect system for
atomicity. In PLDI ’05: Proceedings of the 2005 ACM SIGPLAN
conference on programming language design and implementation,

University Computer Laboratory, 2003. Also available as Technical

[8] Tim Harris. Exceptions and side-effects in atomic blocks. In
Proceedings of the 2004 Workshop on Concurrency and Synchronization
Memorial University of Newfoundland CS Technical Report 2004-01.

[9] Tim Harris and Keir Fraser. Language support for lightweight
transactions. In Object-Oriented Programming, Systems, Languages,

[10] Tim Harris and Keir Fraser. Revocable locks for non-blocking
programming. In Symposium on Principles and Practice of Parallel

Scherer III. Software transactional memory for dynamic-sized data

Architectural support for lock-free data structures. In Proceedings of
the 20th Annual International Symposium on Computer Architecture,

and list-based sets. In SPAA ’02: Proceedings of the fourteenth annual
ACM symposium on Parallel algorithms and architectures, pages 73–82,

Computer Sciences, University of Wisconsin, pages 1–11, Mar 2005.

Weimer. Cil: Intermediate language and tools for analysis and
transformation of c programs. In CC ’02: Proceedings of the 11th
International Conference on Compiler Construction, pages 213–228,


[17] Ravi Rajwar and James R. Goodman. Transactional lock-free
execution of lock-based programs. In Proceedings of the 10th
Symposium on Architectural Support for Programming Languages

[18] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing trans-
actional memory. In Proceedings of the 32nd Annual International
Symposium on Computer Architecture, pages 494–505. IEEE Compu-

for multithreaded programs. In Symposium on Principles and Practice

Eraser: A dynamic data race detector for multi-threaded programs.

of PLDI 1989.

[22] Ori Shalev and Nir Shavit. Split-ordered lists: lock-free extensible
hash tables. In PODC ’03: Proceedings of the twenty-second annual
symposium on Principles of distributed computing, pages 102–111,

[23] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula,
and Eric Brewer. Capriccio: scalable threads for internet services.
In SOSP ’03: Proceedings of the nineteenth ACM symposium on
Operating systems principles, 2003.

of data race conditions via adaptive tracking. Technical report,