Lock Inference for Atomic Sections

Michael Hicks
University of Maryland, College Park
mwh@cs.umd.edu

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

Polyvios Pratikakis
University of Maryland, College Park
polyvios@cs.umd.edu

Abstract

To prevent unwanted interactions in multithreaded programs, programmers have traditionally employed pessimistic, blocking concurrency primitives. Using such primitives correctly and efficiently is notoriously difficult. To simplify the problem, recent research proposes that programmers specify *atomic sections* of code whose executions should be atomic with respect to one another, without dictating exactly how atomicity enforced. Much work has explored using optimistic concurrency, or *software transactions*, as a means to implement atomic sections.

This paper proposes to implement atomic sections using a static whole-program analysis to insert necessary uses of pessimistic concurrency primitives. Given a program that contains programmer-specified atomic sections and thread creations, our *mutex inference algorithm* efficiently infers a set of locks for each atomic section that should be acquired (released) upon entering (exiting) the atomic section. The key part of this algorithm is determining which memory locations in the program could be shared between threads, and using this information to generate the necessary locks. To determine sharing, our analysis uses the notion of *continuation effects* to track the locations accessed after each program point. As continuation effects are flow sensitive, a memory location may be thread-local before a thread creation and thread-shared afterward. We prove that our algorithm is correct, and provides parallelism according to the precision of the points-to analysis. While our algorithm also attempts to reduce the number locks while preserving parallelism, we show that minimizing the number of locks is NP-hard.

1. Introduction

Concurrent programs strive to balance *safety* and *liveness*. Programmers typically ensure safety by, among other things, using blocking synchronization primitives such as mutual exclusion locks to restrict concurrent accesses to data. Programmers ensure liveness by reducing waiting and blocking as much as possible, for example by using more mutual exclusion locks at a finer granularity. Thus these two properties are in tension: ensuring safety can result in reduced or no parallelism, compromising liveness, while ensuring liveness could permit concurrent access to an object (a data race) potentially compromising safety. Balancing this tension manually can be quite difficult[1], particularly since traditional uses of blocking synchronization are not modular, and thus the programmer must reason about the entire program's behavior.

Software *transactions* promise to improve this situation. A transaction is a programmer-designated section of code that should be serializable, so that its execution appears to be atomic[2] with respect to all other transactions in the program. Assuming all concurrently-shared data is accessed within atomic sections, the compiler and runtime system guarantee freedom from data races and deadlocks automatically. Thus, transactions are composable—they can be reasoned about in isolation, without worry that an ill-fated combination of atomic sections could deadlock. This characteristic clearly makes transactions easier to use than having to manipulate low-level mutexes directly in the program.

Recent research proposes implementing atomic sections using optimistic concurrency techniques [5, 6, 7, 12, 13]. Roughly speaking, memory accesses within a transaction are logged. At the conclusion of the transaction, if the log is consistent with the current state of memory, then the writes are committed; if not, the transaction is rolled back and restarted. The main drawbacks with this approach are that first, it does not interact well with I/O, which cannot always be rolled back; second, performance can be worse than traditional pessimistic techniques due to the costs of logging and rollback [9].

In this paper, we explore the use of pessimistic synchronization techniques to implement atomic sections. We assume that a program contains occurrences of `fork` e for creating multiple threads and programmer-annotated atomic sections `atomic` e for protecting shared data. For such a program, our algorithm automatically constructs a set of locks and inserts the necessary lock acquires and releases before and after the body of each marked atomic section. A trivial implementation would be to begin and end all atomic sections by, respectively, acquiring and releasing a single global lock. However, an important goal of our algorithm is to maximize parallelism. We present an improved algorithm that uses much finer locking but still enforces atomicity, without introducing deadlock. We implement this algorithm in a tool called LOCKPICK, using the sharedness analysis performed by our race detection tool for C programs, LOCKSMITH [10]. We present an overview of our algorithm next, and describe it in detail in the rest of the paper.

1.1 Overview

The main idea of our approach is simple. We begin by performing a points-to analysis on the program, which maps each pointer in the program to an abstract name that represents the memory pointed to at run time. Then we can create one mutual exclusion lock for each abstract name from the points-to analysis and use it to guard accesses to the corresponding run-time memory locations. At the start of each atomic section, the compiler inserts code to acquire all locks that correspond to the abstract locations accessed within the atomic section. The locks are released when the section concludes. To avoid deadlock, locks are always acquired according to a statically-assigned total order. Since atomic sections might be nested, locks must also be reentrant. Moreover, locations accessed

[1] As of the time this paper is written, Google returns 13,000 pdf documents containing the phrase "notoriously difficult", the word "software", and one of the words "multithreaded" or "concurrent."

[2] For the remainder of the paper, we use the term "atomic" liberally, to mean "appears to be atomic," or "serializable."

$$\begin{array}{llcl}
\text{expressions} & e & ::= & x \mid v \mid e_1\ e_2 \mid \texttt{ref}\ e \mid \texttt{!}\,e \mid e_1 := e_2 \\
 & & \mid & \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & & \mid & \texttt{fork}^i\ e \mid \texttt{atomic}^i\ e \\
\text{values} & v & ::= & n \mid \lambda x.e \\
\text{types} & \tau & ::= & \mathit{int} \mid \mathit{ref}^{\,\rho}\ \tau \mid (\tau,\varepsilon) \to^{\chi} (\tau',\varepsilon') \\
\text{labels} & l & ::= & \rho \mid \varepsilon \mid \chi \\
\text{constraints} & C & ::= & \emptyset \mid \{l \le l'\} \mid C \cup C
\end{array}$$

Figure 1. Source Language, Types, and Constraints

within an inner section are considered accessed in its surrounding sections, to ensure that the global order is preserved.

This approach ensures that no locations are accessed without holding their associated lock. Moreover, locks are not released during execution of an atomic section, and hence all accesses to locations within that section will be atomic with respect to other atomic sections [4]. Our algorithm assumes that shared locations are only accessed within atomic sections; this can be enforced with a small modification of our algorithm, or by using a race detection tool such as LOCKSMITH as a post-pass.

Our algorithm performs two optimizations over the basic approach. First, we reduce our consideration to only those abstract locations that may be shared between threads, since thread-local locations need not be protected by synchronization. Second, we observe that some locks may be coalesced. In particular, if lock ℓ is always held with lock ℓ', then lock ℓ' can safely be discarded.

We implement this approach in two main steps. First, we use a context-sensitive points-to and effect analysis to determine the shared abstract locations as well as the locations accessed within an atomic section (Section 2.2). The points-to analysis is flow-insensitive, but the effect analysis calculates per-program point *continuation effects* that track the effect of the continuation of an expression. Continuation effects let us model that only locations that are used *after* a call to `fork` are shared. The sharing analysis presented here is essentially unchanged from LOCKSMITH's sharing analysis (with only the exception of context sensitivity for simplicity), which has not been presented formally before.

Second, given the set of shared locations, we perform *mutex inference* to determine an appropriate set of locks to guard accesses to the shared locations (Section 3). This phase includes a straightforward algorithm that performs mutex coalescence, to reduce the number of locks while retaining the maximal amount of parallelism. Our algorithm starts by assuming one lock per shared location and iteratively coarsens this assignment, dropping unneeded locks. The algorithm runs in time $O(mn^2)$, where n is the number of shared locations in the program and m is the number of atomic sections. We show that the resulting locking discipline provides exactly the same amount of parallelism as the original, non-coalesced locking discipline, while at the same time uses fewer locks. Our algorithm is not optimal, because it does not always reach the minimum number of locks possible. Indeed, in section 3.2 we prove that using the minimum number of locks is an NP-hard problem.

2. Shared Location Inference

Figure 1 shows the source language we use to illustrate our inference system. Our language is a lambda calculus extended with integers, comparisons, updatable references, thread creation $\texttt{fork}^i\ e$, and atomic sections $\texttt{atomic}^i\ e$; in the latter two cases the i is an index used to refer to the analysis results. The expression $\texttt{fork}^i\ e$ creates a new child thread that evaluates e and discards the result, continuing with normal evaluation in the parent thread. Our approach can easily be extended to support polymorphism and polymorphic recursion for labels in a standard way [11], as LOCKSMITH does [10], but we omit rules for polymorphism because they add complication but no important issues.

We use a type-based analysis to determine the set of abstract locations ρ, created by `ref`, that could be shared between threads in some program e. We compute this using a modified *label flow analysis* [10, 11]. Our system uses three kinds of labels: *location labels* ρ, *effects* χ and *continuation effects* ε. Effects of both kinds represent those locations ρ dereferenced or assigned to during a computation. Typing a program generates *label flow constraints* of the form $l \le l'$. Afterwards, these constraints are solved to learn the desired information. The constraint $l \le l'$ is read "label l flows to label l'." For example, if `x` has type $\mathit{ref}^{\,\rho}\ \tau$, and we have constraints $\rho' \le \rho$ and $\rho'' \le \rho$, then `x` may point to the locations ρ' or ρ''. Labels also flow to effects χ or ε, so for example if $\rho \le \chi$ then an expression with effect χ may access location ρ.

The typing judgment has the following form:

$$C; \varepsilon; \Gamma \vdash e : \tau^{\chi}; \varepsilon'$$

This means that in type environment Γ, expression e has *effect type* τ^{χ} given constraints C. Effect types τ^{χ} consist of a type τ annotated with the effect χ of e. Within the type rules, the judgment $C \vdash l \le l'$ indicates that $l \le l'$ can be proven by the constraint set C. In an implementation, such judgments cause us to generate constraint $l \le l'$ and add it C. Types include standard integer types; updatable reference types $\mathit{ref}^{\,\rho}\ \tau$, each of which is decorated with a location label ρ; and function types of the form $(\tau,\varepsilon) \to^{\chi} (\tau',\varepsilon')$, where τ and τ' are the domain and range types, and χ is the effect of calling the function. We explain ε' and ε on function types momentarily.

The judgment $C; \varepsilon; \Gamma \vdash e : \tau^{\chi}; \varepsilon'$ is standard for effect inference except for ε and ε', which express *continuation effects*. Here, ε is the *input effect*, which denotes locations that may be accessed *during* or *after* evaluation of e. The *output effect* ε' contains locations that may be accessed *after* evaluation of e (thus all locations in ε' will be in ε). We use continuation effects in the rule for `fork` e to determine sharing. In particular, we infer that a location is shared if it is in the input effect of the child thread and the output effect of the `fork` (and thus may be accessed subsequently in the parent thread).

In addition to continuation effects ε, we also compute the effects χ of a lexical expression, stored as an annotation on the expression's type. We use effects χ to compute all dereferences and assignments that occur within the body of an atomic transaction. We cannot simply use continuation effects ε, since those also include all dereferences that happen in the continuation of the program after the atomic section. Note that we cannot compute standard effects given continuation effects ε. The effect of an expression e is not simply its input continuation effect minus the output continuation effect, since that could remove locations accessed both within e and after it.

Returning to the explanation of function types, the effect label ε' denotes the set of locations accessed after the function returns, while ε denotes those locations accessed after the function is called, including any locations in ε'.

Example Consider the following program:

```
let x = ref 0 in
let y = ref 1 in
  x := 4;
  fork^1 (!x; !y);
  /* (1) */
  y := 5
```

In this program two variables x and y refer to memory locations. x is initialized and updated, but then is handed off to the child thread and no longer used by the parent thread. Hence x can be treated as thread-local. On the other hand, y is used both by the parent and child thread, and hence must be modeled as shared.

Because we use continuation effects, we model this situation precisely. In particular, the input effect of the child thread is $\{x, y\}$. The output effect of the fork (i.e. starting at (1)) is $\{y\}$. Since $\{x, y\} \cap \{y\} = \{y\}$, we determine that only y is shared. If instead we had used regular effects, and we simply intersected the effect of the parent thread with the child thread, we would think that x was shared even though it is handed off and never used again by the parent thread.

Moreover, the system that we present in this paper does not differentiate between read and write accesses, hence it will infer that read-only variables are shared. In practice, we wish to allow read-only values to be accessed freely by all threads. To do that, we differentiate between read and write effects, and do not consider values that only appear in the read effects of both threads to be shared.

2.1 Type Rules

Figure 2 gives the type inference rules for sharing inference. We discuss the rules briefly. [Id] and [Int] are straightforward. Notice that since neither accesses any locations, the input and output effects are the same, and their effect χ is unconstrained (and hence will be empty during constraint resolution). In [Lam], the labels ε_{in} and ε_{out} that are bound in the type correspond to the input and output effects of the function. Notice that the input and output effects of $\lambda x.e$ are both just ε, since the definition itself does not access any locations—the code in e will only be evaluated when the function is applied. Finally, the effect χ of the function is drawn from the effect of e.

In [App], the output effect ε_1 of evaluating e_1 becomes the input effect of evaluating e_2. This implies a left-to-right order of evaluation: Any locations that may be accessed during or after evaluating e_2 also may be accessed after evaluating e_1. The function is invoked after e_2 is evaluated, and hence e_2's output effect must be ε_{in} from the function signature. [Sub], described below, can always be used to achieve this. Finally, notice that the effect of the application is the effect χ of evaluating e_1, evaluating e_2, and calling the function. [Sub] can be used to make these effects the same.

[Cond] is similar to [App], where one of e_1 or e_2 is evaluated after e_0. We require both branches to have the same output effect ε' and regular effect χ, and again we can use [Sub] to achieve this.

[Ref] creates and initializes a fresh location but does not have any effect itself. This is safe because we know that location ρ cannot possibly be shared yet.

[Deref] accesses location ρ after e is evaluated, and hence we require that ρ is in the continuation effect ε' of e, expressed by the judgment $C \vdash \rho \le \varepsilon'$. In addition, we require that the dereferenced location is in the effects $\rho \le \chi$. Note that [Sub] can be applied before applying [Deref] so that this does not constrain the effect of e. The rule for [Assign] is similar. Notice that the output effect of $!\,e$ is the same the effect ε' of e. This is conservative because ρ must be included in ε' but may not be accessed again following the evaluation of $!\,e$. However, in this case we can always apply [Sub] to remove it.

$$[\text{Id}]\ \frac{}{C;\varepsilon;\Gamma, x:\tau \vdash x : \tau^{\chi};\varepsilon}$$

$$[\text{Int}]\ \frac{}{C;\varepsilon;\Gamma \vdash n : \mathit{int}^{\chi};\varepsilon}$$

$$[\text{Lam}]\ \frac{C;\varepsilon_{in};\Gamma, x:\tau_{in} \vdash e : \tau_{out}^{\chi};\varepsilon_{out}}{C;\varepsilon;\Gamma \vdash \lambda x.e : (\tau_{in},\varepsilon_{in}) \to^{\chi} (\tau_{out},\varepsilon_{out});\varepsilon}$$

$$[\text{App}]\ \frac{\begin{array}{c} C;\varepsilon;\Gamma \vdash e_1 : \tau_{fun}^{\chi};\varepsilon_1 \\ \tau_{fun} = (\tau_{in},\varepsilon_{in}) \to^{\chi} (\tau_{out},\varepsilon_{out}) \\ C;\varepsilon_1;\Gamma \vdash e_2 : \tau_{in}^{\chi};\varepsilon_{in} \end{array}}{C;\varepsilon;\Gamma \vdash e_1\ e_2 : \tau_{out}^{\chi};\varepsilon_{out}}$$

$$[\text{Cond}]\ \frac{\begin{array}{c} C;\varepsilon;\Gamma \vdash e_0 : \mathit{int}^{\chi};\varepsilon_0 \\ C;\varepsilon_0;\Gamma \vdash e_1 : \tau^{\chi};\varepsilon' \\ C;\varepsilon_0;\Gamma \vdash e_2 : \tau^{\chi};\varepsilon' \end{array}}{C;\varepsilon;\Gamma \vdash \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau^{\chi};\varepsilon'}$$

$$[\text{Ref}]\ \frac{C;\varepsilon;\Gamma \vdash e : \tau^{\chi};\varepsilon'}{C;\varepsilon;\Gamma \vdash \texttt{ref}\ e : (\mathit{ref}^{\,\rho}\ \tau)^{\chi};\varepsilon'}$$

$$[\text{Deref}]\ \frac{\begin{array}{c} C;\varepsilon;\Gamma \vdash e : (\mathit{ref}^{\,\rho}\ \tau)^{\chi};\varepsilon' \\ C \vdash \rho \le \varepsilon' \quad C \vdash \rho \le \chi \end{array}}{C;\varepsilon;\Gamma \vdash !\,e : \tau^{\chi};\varepsilon'}$$

$$[\text{Assign}]\ \frac{\begin{array}{c} C;\varepsilon;\Gamma \vdash e_1 : (\mathit{ref}^{\,\rho}\ \tau)^{\chi};\varepsilon_1 \\ C;\varepsilon_1;\Gamma \vdash e_2 : \tau^{\chi};\varepsilon_2 \\ C \vdash \rho \le \varepsilon_2 \quad C \vdash \rho \le \chi \end{array}}{C;\varepsilon;\Gamma \vdash e_1 := e_2 : \tau^{\chi};\varepsilon_2}$$

$$[\text{Sub}]\ \frac{\begin{array}{c} C;\varepsilon;\Gamma \vdash e : \tau^{\chi};\varepsilon' \\ C \vdash \tau \le \tau_1 \quad C \vdash \chi \le \chi_1 \quad C \vdash \varepsilon'' \le \varepsilon' \end{array}}{C;\varepsilon;\Gamma \vdash e : \tau_1^{\chi_1};\varepsilon''}$$

$$[\text{Fork}]\ \frac{\begin{array}{c} C;\varepsilon_e^i;\Gamma \vdash e : \tau^{\chi};\varepsilon_e' \\ C \vdash \varepsilon_e^i \le \varepsilon \quad C \vdash \varepsilon^i \le \varepsilon \end{array}}{C;\varepsilon;\Gamma \vdash \texttt{fork}^i\ e : \mathit{int}^{\chi'};\varepsilon^i}$$

$$[\text{Atomic}]\ \frac{C;\varepsilon;\Gamma \vdash e : \tau^{\chi^i};\varepsilon'}{C;\varepsilon;\Gamma \vdash \texttt{atomic}^i\ e : \tau^{\chi^i};\varepsilon'}$$

Figure 2. Type Inference Rules

[Sub] introduces sub-effecting to the system. In this rule, we implicitly allow χ_1 and ε'' to be fresh labels. In this way we can always match the effects of subexpressions, e.g., of e_1 and e_2 in [Assign], by creating a fresh variable χ and letting $\chi_1 \le \chi$ and $\chi_2 \le \chi$ by [Sub], where χ_1 and χ_2 are effects of e_1 and e_2. Notice that subsumption on continuation effects is contravariant: whatever output effect ε'' we give to e, it must be included in its original effect ε'. [Sub] also introduces subtyping via the judgment $C \vdash \tau \le \tau'$, as shown in Figure 3. The subtyping rules are standard except for the addition of effects in [Sub-Fun]. Continuation effects are contravariant to the direction of flow of regular types, similarly to the output effects in [Sub].

[Fork] models thread creation. The regular effect χ' of the fork is unconstrained, since in the parent thread there is no effect. The continuation effect ε_e^i captures the effect of the child thread evaluating e, and the effect ε^i captures the effect of the rest of the parent thread's evaluation. To infer sharing (discussed in section

$$\text{[Sub-Int]}\frac{}{C \vdash \mathit{int} \leq \mathit{int}}$$

$$\text{[Sub-Ref]}\frac{C \vdash \rho_1 \leq \rho_2 \quad C \vdash \tau_1 \leq \tau_2 \quad C \vdash \tau_2 \leq \tau_1}{C \vdash \mathit{ref}^{\rho_1}\ \tau_1 \leq \mathit{ref}^{\rho_2}\ \tau_2}$$

$$\text{[Sub-Fun]}\frac{\begin{array}{cc} C \vdash \tau_2 \leq \tau_1 & C \vdash \tau_1' \leq \tau_2' \\ C \vdash \varepsilon_1 \leq \varepsilon_2 \quad C \vdash \varepsilon_2' \leq \varepsilon_1' & C \vdash \chi_1 \leq \chi_2 \end{array}}{C \vdash (\tau_1, \varepsilon_1) \to^{\chi_1} (\tau_1', \varepsilon_1') \leq (\tau_2, \varepsilon_2) \to^{\chi_2} (\tau_2', \varepsilon_2')}$$

Figure 3. Subtyping Rules

2.2) we will compute $\varepsilon_e^i \cap \varepsilon^i$; this is the set of locations that could be accessed by both the parent and child thread after the fork. Notice that the input effect ε_e^i of the child thread is included in the input effect of the `fork` itself. This effectively causes a parent to "inherit" its child's effects, which is important for capturing sharing between two child threads. Consider, for example, the following program:

```
let x = ref 0 in
  fork¹ (!x);
  /* (1) */
  fork² (x := 2)
```

Notice that while x is created in the parent thread, it is only accessed in the two child threads. Let ρ be the location of x. Then ρ is included in the continuation effect at point (1), because the effect of the child thread $\texttt{fork}^2\ x := 2$ is included in the effect of the call at (1). Thus when we compute the intersection of the input effect of $\texttt{fork}^1\ !x$ with the output effect of the parent (which starts at (1)), the result will contain ρ, which we will hence determine to be shared.

Finally, [Atomic] models atomic sections, which have no effect on sharing. During mutex inference, we will use the solution to the effect χ^i of each atomic section to infer the needed locks. Notice that the effect of $\texttt{atomic}^i\ e$ is the same as the effect of e; this will ensure that atomic sections compose properly and not introduce deadlock.

Soundness Standard label flow and effect inference has been shown to be sound [8, 11], including polymorphic label flow inference. We believe it is straightforward to show that continuation effects are a sound approximation of the locations accessed by the continuation of an expression.

2.2 Computing Sharing

Similarly to standard type-based label flow analysis, we apply the type inference rules in Figures 2 and 3, which produce a set of label flow constraints C. One can think of these constraints as forming a directed graph, where each label forms a node and every constraint $l \leq l'$ is represented as a directed edge from l to l'. Then for each label l, we compute the set $S(l)$ of location labels ρ that "flow" to l by transitively closing the graph. The total time to transitively close the graph is $O(n^2)$, where n is the number of nodes in the graph. (Given a polymorphic inference system, we could compute label flow using context-free language reachability in time cubic in the size of the type-annotated program).

Unlike standard type-based label flow analysis, our label flow graph includes labels ε to encode continuation effects. Recall that we define input and output continuation effects $\varepsilon, \varepsilon'$ for every expression e in the program. In the solved points-to graph, the flow solutions of $\varepsilon, \varepsilon'$ include all location labels that are accessed by the continuation of the program after the expression e; the solution of ε moreover includes the effect of e.

Once we have computed $S(\varepsilon)$ for all effect labels ε, we visit each $\texttt{fork}^i$ in the program. Then the set of shared locations for the program *shared* is given by

$$\mathit{shared} = \bigcup_i (S(\varepsilon^i) \cap S({\varepsilon_e}^i))$$

In other words, any locations accessed in the continuation of a parent and its child threads at a `fork` are shared.

3. Mutex Inference

Given the set of shared locations, the next step is to compute a set of locks used to guard all of the shared locations. A simple and correct solution is to associate a lock ℓ_ρ with each shared location $\rho \in \mathit{shared}$. Then at the beginning to a section $\texttt{atomic}^i\ e$, we acquire all locks associated with locations in χ^i. To prevent deadlock, we also impose a total ordering on all the locks, acquiring the locks in that order.

This approach is sound and in general allows more parallelism than the naïve approach of using a single lock for all atomic sections.[3] However, a program of size n may have $O(n)$ locations, and acquiring that many locks would introduce unwanted overhead, particularly on a multi-processor machine. Thus we would like to use fewer locks while maintaining the same level of parallelism. Computing a minimum set of locks is NP-hard, as shown in section 3.2. We propose an efficient but non-optimal algorithm based on the following observation: if two locations are always accessed together, then they can be protected by the same mutex without any loss of parallelism.

DEFINITION 1 (Dominates). *We say that accesses to location ρ* dominate *accesses to location ρ', written $\rho \geq \rho'$, if every atomic section containing an access to ρ' also contains an access to ρ.*

We write $\rho > \rho'$ for strict domination, i.e., $\rho \geq \rho'$ and $\rho \neq \rho'$. Thus, whenever $\rho > \rho'$ we can use ρ's mutex to protect both ρ and ρ'. Notice that the dominates relationship is not symmetric. For example, we might have a program containing two atomic sections, $\texttt{atomic}\ (!x; !y)$ and $\texttt{atomic}\ !x$. In this program, the location of x dominates the location of y but not vice-versa. Domination is transitive, however.

Computing the dominates relationship is straightforward. For each location ρ, we initially assume $\rho > \rho'$ for all locations ρ'. Then for each $\texttt{atomic}^i\ e$ in the program, if $\rho' \in S(\chi^i)$ but $\rho \notin S(\chi^i)$, then we remove our assumption $\rho > \rho'$. This takes time $O(m|\mathit{shared}|)$ for each ρ, where m is the number of atomic sections. Thus in total this takes time $O(m|\mathit{shared}|^2)$ for all locations.

Given the dominates relationship, we then compute a set of locks to guard shared locations using the following algorithm:

ALGORITHM 2 (Mutex Selection). *Computes a mapping $L : \rho \to \ell$ from locations ρ to lock names ℓ. We call L a* mutex selection *function.*

```
1. For each $\rho \in shared$, set $L(\rho) = \ell_\rho$
2. For each $\rho \in shared$
3.     If there exists $\rho' > \rho$, then
4.         For each $\rho''$ such that $L(\rho'') = \ell_\rho$
5.             $L(\rho'') := \ell_{\rho'}$
```

[3] If we had a more discerning points-to analysis, or if we acquired the locks piecemeal within the atomic section, rather than all at the start [9], we would do even better. We consider this issue at the end of the next section.

In each step of the algorithm, we pick a location ρ and replace all occurrences of its lock by a lock of any of its dominators. Notice that the order in which we visit the set of locks is unspecified, as is the particular dominator to pick. We prove below that this algorithm maintains maximum parallelism, no matter the ordering. Mutex selection takes time $O(|\mathit{shared}|^2)$, since for each location ρ we must examine L for every other shared location.

The combination of computing the dominates relationship and mutex selection yields mutex inference. We pick a total ordering on all the locks in *range*(L). Then we replace each $\texttt{atomic}^i\ e$ in the program with code that first acquires all the locks in $L(S(\chi^i))$ in order, performs the actions in e, and then releases all the locks. Put together, computing the dominates relationship and mutex selection takes $O(m|\mathit{shared}|^2)$ time.

Examples To illustrate the algorithm, consider the set of accesses of the atomic sections in the program. For clarity we simply list the accesses, using English letters to stand for locations. For illustration purposes we also assume all locations are shared. For a first example, suppose there are three atomic sections with the following pattern of accesses

$$\{a\} \qquad \{a, b\} \qquad \{a, b, c\}$$

Then we have $a > b$, $a > c$, and $b > c$. Initially $L(a) = \ell_a$, $L(b) = \ell_b$, and $L(c) = \ell_c$. Suppose in the first iteration of the algorithm location c is chosen, and we pick $b > c$ as the dominates relationship to use. Then after one iteration, we will have $L(c) = \ell_b$. On a subsequent iteration, we will eventually pick location b with $a > b$, and set $L(b) = L(c) = L(a) = \ell_a$. It is easy to see that this same solution will be computed no matter the choices made by the algorithm. And this solution is what we want: Since b and c are always accessed along with a, we can eliminate b's lock and c's lock.

As another example, suppose we have the following access pattern:

$$\{a\} \qquad \{a, b, c\} \qquad \{b\}$$

Then we have $a > c$ and $b > c$. The only interesting step of the algorithm is when it visits node c. In this case, the algorithm can either set $L(c) = \ell_a$ or $L(c) = \ell_b$. However, ℓ_a and ℓ_b are still kept disjoint. Hence upon entering the left-most section ℓ_a is acquired, and upon entering the right-most section ℓ_b is acquired. Thus the left- and right-most sections can run concurrently with each other. Upon entering the middle section we must acquire both ℓ_a and ℓ_b—and hence no matter what choice the algorithm made for $L(c)$, the lock guarding it will be held.

This second example shows why we do not use a naïve approach such as unifying the locks of all locations accessed within an atomic section. If we did so here and we would choose $L(a) = L(b) = L(c)$. This answer would be safe but we could not concurrently execute the left-most and right-most sections.

3.1 Correctness

First, we formalize the problem of mutex inference with respect to the points-to analysis, and prove that our mutex inference algorithm produces a correct solution. Let $S_i = S(\chi^i)$, where χ^i is the effect of atomic section $\texttt{atomic}^i\ e$.

DEFINITION 3 (Parallelism). *The parallelism of a program is a set*

$$\mathcal{P} = \{(i, j) \mid S_i \cap S_j = \emptyset\}$$

In other words, the parallelism of a program is the set of all pairs of atomic sections that could safely execute in parallel, because they access no common locations.

We define the parallelism allowed by a given mutex selection function L similarly, where we overload the meaning of L to apply to sets of locations and return sets of mutexes: $L(S_i) = \{L(\rho) \mid \rho \in S_i\}$.

DEFINITION 4 (Parallelism of L). *The* parallelism *of a mutex selection function $L : \rho \to \ell$, written $P(L)$, is defined as*

$$P(L) = \{(i, j) \mid L(S_i) \cap L(S_j) = \emptyset\}$$

The parallelism $P(L)$ is the set of all possible pairs of atomic sections that could execute in parallel because they have no common associated locks. Let L be the mutex selection function calculated by our algorithm. The objective of mutex inference is to compute a solution L that allows the maximum parallelism possible without breaking atomicity.

LEMMA 1. *If $L(\rho) = \ell_{\rho'}$, then $\rho' \geq \rho$.*

PROOF. We prove this by induction on the number of iterations of step 2 of the algorithm. Clearly this holds for the initial mutex selection function $L_0(\rho) = \ell_\rho$, where we mark the function L that the algorithm has computed so far, with a subscript denoting the current iteration. Then suppose it holds for L_k, the selection function after k iterations of step 2. For an arbitrary $\rho_1 \in$ *shared*, there are two cases:

1. If $L_k(\rho_1) = \ell_\rho$ then $L_{k+1}(\rho_1) = \ell_{\rho'}$. By induction $\rho \geq \rho_1$, and since $\rho' > \rho$ by assumption, we have $\rho' \geq \rho_1$ by transitivity.
2. Otherwise, there exists some ρ_2 such that $L_k(\rho_1) = L_{k+1}(\rho_1) = \ell_{\rho_2}$, and hence by induction $\rho_2 \geq \rho_1$.

$\square$

LEMMA 2 (Correctness). *If L is the mutex selection function computed by the above algorithm, then $P(L) = \mathcal{P}$.*

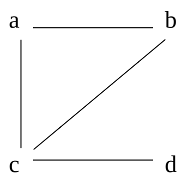
In other words, the algorithm will not let more sections execute in parallel than allowed, and it allows as much parallelism as the uncoalesced, one-lock-per-location approach.

PROOF. We prove this by induction on the number of iterations of step 2 of the algorithm. For the base case, the initial mutex selection function $L_0(\rho) = \ell_\rho$ clearly satisfies this property, because there is a one-to-one mapping between each location and each lock. For the induction step, assume $\mathcal{P} = P(L_k)$ and for step 2 we have $\rho' > \rho$. Let L_{k+1} be the mutex selection function after this step. Pick any i and j. Then there are two directions to show.

$P(L_{k+1}) \subseteq \mathcal{P}$ Assume this is not the case. Then there exist i, j such that $(i, j) \in P(L_{k+1})$ and $(i, j) \notin \mathcal{P}$. From the latter we get $S_i \cap S_j \neq \emptyset$. Then clearly there exists a $\rho'' \in S_i \cap S_j$, and since L_{k+1} is a total function, there must exist an ℓ such that $L_{k+1}(\rho'') = \ell$. But then $(i, j) \notin P(L_{k+1})$ since $L_{k+1}(S_i) \cap L_{k+1}(S_j) \neq \emptyset$. Therefore $P(L_{k+1}) \subseteq \mathcal{P}$.

$P(L_{k+1}) \supseteq \mathcal{P}$ Assume this is not the case. Then there exist i, j such that $(i, j) \notin P(L_{k+1})$ and $(i, j) \in \mathcal{P}$. From the latter we get $S_i \cap S_j = \emptyset$. Also, from the induction hypothesis $L_k(S_i) \cap L_k(S_j) = \emptyset$, and we have $L_{k+1}(S_i) = L_k(S_i)[\ell_\rho \mapsto \ell_{\rho'}]$, and similarly for $L_{k+1}(S_j)$. Suppose that $\ell_\rho \notin L_k(S_i)$ and $\ell_\rho \notin L_k(S_j)$. Then clearly $L_{k+1}(S_i) \cap L_{k+1}(S_j) = \emptyset$, which contradicts $(i, j) \notin P(L_{k+1})$.

Otherwise suppose without loss of generality that $\ell_\rho \in L_k(S_i)$. Then by assumption $\ell_\rho \notin L_k(S_j)$. So clearly the renaming $[\ell_\rho \mapsto \ell_{\rho'}]$ cannot add $\ell_{\rho'}$ to $L_{k+1}(S_j)$. Thus in order to show $L_{k+1}(S_i) \cap L_{k+1}(S_j) = \emptyset$, we need to show $\ell_{\rho'} \notin L_k(S_j)$. Since $\ell_\rho \in L_k(S_i)$, we know there exists a $\rho'' \in S_i$ such that $L_k(\rho'') = \ell_\rho$, which by Lemma 1 implies $\rho \geq \rho''$. But then from $\rho' > \rho$ we have $\rho' \in S_i$. Also, since $S_i \cap S_j = \emptyset$, we have $\rho' \notin S_j$. So suppose for a contradiction that $\ell_{\rho'} \in L_k(S_j)$. Then there must be a $\rho''' \in S_j$



(a) A simple graph.

$\texttt{atomic}^a\ \{x_{ab} := 1;\ x_{ac} := 2\}$
$\texttt{atomic}^b\ \{x_{ab} := 3;\ x_{ac} := 4\}$
$\texttt{atomic}^c\ \{x_{ac} := 6;\ x_{bc} := 7;\ x_{cd} := 5\}$
$\texttt{atomic}^d\ \{x_{cd} := 8\}$

(b) The corresponding atomic transactions.

Figure 4. Reduction Example

such that $L_k(\rho''') = \ell_{\rho'}$. But then by Lemma 1, we have $\rho' \geq \rho'''$. Then $\rho' \in S_j$, a contradiction. Hence we must have $\ell_{\rho'} \notin L_k(S_j)$, and therefore $L_{k+1}(S_i) \cap L_{k+1}(S_j) = \emptyset$, which again contradicts $(i,j) \notin P(L_{k+1})$. Therefore $P(L_{k+1}) \supseteq \mathcal{P}$. □

3.2 NP-Hardness

Although our algorithm maintains the maximum amount of parallelism, it may use more than the minimum number of locks. Ideally, we would like to solve the following problem:

DEFINITION 5 (k-Mutex Inference). *Given a parallel program e and an integer k, is there a mutex selection function L for which $|\mathrm{range}(L)| = k$ and $P(L) = \mathcal{P}$?*

From this, we can state the minimum mutex inference problem.

DEFINITION 6 (Minimum Mutex Inference). *Given a parallel program e, find the minimum is k for which there a mutex selection function L having $|\mathrm{range}(L)| = k$ and $P(L) = \mathcal{P}$.*

However, it turns out that the above problem is NP-hard. We prove this by reducing *minimum edge clique cover* to the mutex inference problem.

DEFINITION 7 (Edge Clique Cover of size k). *Given a graph $G = (V,E)$, and a number k, is there a set of cliques $W_1, \ldots, W_k \subseteq V$ such that for every edge $(v,v') \in E$, there exists some W_i that contains both v and v'?*

DEFINITION 8 (Minimum Edge Clique Cover). *Given a graph $G = (V,E)$, find the minimum k for which there is an edge clique cover of size k for G.*

LEMMA 3. *Minimum Mutex Inference is NP-hard.*

PROOF. The proof is by reduction from the Minimum Edge Clique Cover problem. Specifically, given a graph $G = (V,E)$, we can construct in polynomial time a program e such that there exists a mutex selection function L for e for which $|\mathrm{range}(L)| = k$ and $P(L) = \mathcal{P}$ if and only if there exists an edge clique cover of size k for G.

The construction algorithm is:

- For every vertex $v_i \in V$, create an atomic transaction α_i.
- For every edge $(v_i, v_j) \in E$, create a fresh global location ρ_{ij}, and add a dereference of ρ_{ij} in the body of both α_i and α_j.

Note that the only location that can be accessed in both of two atomic transactions α_i and α_j is ρ_{ij}, since there can be only one edge between v_i and v_j. Figure 4(b) shows the program created for the graph in figure 4(a).

case $\Rightarrow$ Suppose that there exists a selection function L and an integer k, such that $|\mathrm{range}(L)| = k$. Then we can construct an edge clique cover $W_1, ..., W_k$ for G, where $W_i \subseteq V$ for $1 \leq i \leq k$. We construct these sets as follows. For every lock $\ell_i \in \mathrm{range}(L)$, we construct the set $W_i \subseteq V$ by adding to W_i all vertices v_j such that $\ell_i \in L(\alpha_j)$. Here by $L(\alpha_j)$ we mean the set of locks computed by applying L to every ρ dereferenced in α_j. To prove $W_1, ..., W_k$ is an edge clique cover, we must show that each W_i is a clique on G, and that all cliques cover E.

The first claim is easily proved by contradiction: assume W_i is not a clique on $G = (V,E)$; then there exists a pair of vertices $v_m, v_n \in W_i$ such that the edge $(v_m, v_n) \notin E$. In that case, there is no location ρ_{mn} created by the reduction algorithm that is accessed in both α_m and α_n. In that case, we have by definition that $(m,n) \in \mathcal{P}$, i.e., α_m and α_n can be executed in parallel. But, since $v_m, v_n \in W_i$, we get by construction of W_i that there must exist a lock ℓ_i such that $\ell_i \in L(\alpha_m)$ and $\ell_i \in L(\alpha_n)$. This would mean that $(m,n) \notin P(L)$, because both α_m and α_n acquire ℓ_i. Hence, we get $P(L) \neq \mathcal{P}$, a contradiction.

We also claim that the set of cliques $W_i, 1 \leq i < k$ covers all the edges in E. To prove this, assume that it does not: Then there exists an edge $(v_m, v_n) \in E$, but there is no clique W_i covering that edge: i.e., there is no W_i such that $v_m \in W_i$ and $v_n \in W_i$, for $1 \leq i < k$. By construction we have that the location ρ_{mn} is accessed in both atomic transactions α_m and α_n. By the definition of L, there must be a lock ℓ_i such that $L(\rho_{mn}) = \ell_i$. Since both α_m and α_n access ρ_{mn}, the lock ℓ_i is held during both. In that case, there exists a clique W_i that contains both v_m and v_n. This contradicts the assumption, therefore all edges in E are covered by the cliques $W_1, ..., W_k$.

To illustrate, suppose the lock selection function L for the program of Figure 4(b) uses 3 locks to synchronize this program, as follows:

$$L(\rho_{ab}) = \ell_1, \quad L(\rho_{bc}) = \ell_1, \quad L(\rho_{ac}) = \ell_2, \quad L(\rho_{cd}) = \ell_3$$

Then the clique cover we construct for the graph for this mutex selection will include 3 cliques, one per lock in the range of L. W_1 will include all the atomic sections that must acquire ℓ_1, which is a,b and c; W_2 will include a, b, and c and W_3 will include c and d. Together, W_1, W_2, and W_3 form an edge clique cover of size 3.

case $\Leftarrow$ Suppose there exists an edge clique cover $W_1, ..., W_k$ for the graph G. Then we can construct a mutex selection function L for e such that $|\mathrm{range}(L)| = k$ and $P(L) = \mathcal{P}$. We do this as follows. For every clique W_i we create a lock ℓ_i. Then for every $v_m, v_n \in W_i$ we set $L(\rho_{mn}) = \ell_i$.

Clearly, $\mathrm{range(L)} = k$. It remains to show $P(L) = \mathcal{P}$. First, we show $\mathcal{P} \subseteq P(L)$. Let $(m,n) \in \mathcal{P}$, meaning that two atomic blocks α_m and α_n in the constructed program e can run in parallel, or α_m and α_n do not access any variable in common. Therefore, by construction of the program e, graph G cannot include the edge (v_m, v_n). This means that there is no clique W_i containing both v_m and v_n. Then, there is no lock ℓ_i that is held during both α_m and α_n, which gives $(m,n) \in P(L)$. Now we show $P(L) \subseteq \mathcal{P}$. If $(m,n) \in P(L)$ then there is no lock ℓ_i that is held for both α_m and α_n. From the construction of L we get that there is no clique W_i that contains both v_m and v_n, therefore there is no edge in G between v_m and v_n. So, there is no common location ρ_{mn} accessed by α_m and α_n, which means $(m,n) \in \mathcal{P}$.

For example, the graph of Figure 4(a), has a 2-clique cover (which is also the minimum): $W_1 = \{a,b,c\}$ and $W_2 = \{c,d\}$. The corresponding mutex selection for the program in Figure 4(b) would use 2 mutexes; ℓ'_1 to protect x_{ab}, x_{bc} and x_{ac}, and ℓ'_2 to protect x_{cd}.

Finally, the complexity of constructing a mutex inference problem e given a graph $G = (V, E)$ is obviously $O(|V| + |E|)$, and the complexity of constructing an edge clique cover given a mutex selection function L on e is obviously $O(k \cdot |V|)$.

To sum up, we have shown that edge clique cover is polynomially reducible to mutex inference. Since Minimum Edge Clique Cover is NP-hard, we have proved that Minimum Mutex Inference is also NP-hard. $\square$

4. Discussion

One restriction of our analysis is that it always produces a finite set of locks, even though programs may use an unbounded amount of memory. Consider the case of a linked list in which atomic sections only access the data in one node of the list at a time. In this case, we could potentially add per-node locks plus one lock for the list backbone. In our current algorithm, however, since all the lock nodes are aliased, we would instead infer only the list backbone lock and use it to guard all accesses to the nodes. LOCKSMITH [10] provides special support for the per-node lock case by using existential types, and we have found it improves precision in a number of cases. It would be useful to adapt our approach to infer these kinds of locks within data structures. One challenge in this case is maintaining lock ordering, since locks would be dynamically generated. A simple solution would be to use the run-time address of the lock as part of the order.

Our algorithm is correct only if all accesses to shared locations occur within atomic sections [4]. Otherwise, some location could be accessed simultaneously by concurrent threads, creating a data race and violating atomicity. We could address this problem in two ways. The simplest thing to do would be to run LOCKSMITH on the generated code to detect whether any races exist. Alternatively, we could modify the sharing analysis to distinguish two kinds of effects: those within an atomic section, and those outside of one. If some location ρ is in the latter category, and $\rho \in$ *shared*, then we have a potential data race we can signal to the programmer.

Our work is closely related to McCloskey et al's Autolocker [9], which also seeks to use locks to enforce atomic sections. There are two main differences between our work and theirs. First, Autolocker requires programmers to annotate potentially shared data with the lock that guards that location. In our approach, such a lock is inferred automatically. However, in Autolocker, programmers may specify per-node locks, as in the above list example, whereas in our case such fine granularity is not possible. Second, Autolocker may not acquire all locks at the beginning of an atomic section, as we do, but rather delay until the protected data is actually dereferenced for the first time. This admits better parallelism, but makes it harder to ensure the lack of deadlock. Our approaches are complementary: our algorithm could generate the needed locks and annotations, and then use Autolocker for code generation.

Flanagan et al [3] have studied how to infer sections of Java programs that behave atomically, assuming that all synchronization has been inserted manually. Conversely, we assume the programmer designates the atomic section, and we infer the synchronization. Later work by Flanagan and Freund [2] looks at adding missing synchronization operations to eliminate data races or atomicity violations. However, this approach only works when a small number of synchronization operations are missing.

We are in the process of implementing our mutex inference algorithm as part of a tool called LOCKPICK, which inserts locking operations in a given program with marked atomic transactions. LOCKPICK uses the points-to and effect analysis of LOCKSMITH to find all shared locations. The analysis extends the formal system described earlier to include label polymorphism, adding context sensitivity. LOCKPICK uses a C type attribute to mark a function as atomic. For example, in the following code:

```
int foo(int arg) __attribute__((atomic)) {
  // atomic code
}
```

the function `foo` is assumed to contain an atomic section.

We expect LOCKPICK will be a good fit for handling concurrency in Flux [1], a component language for building server applications. Flux defines concurrency at the granularity of individual components, which are essentially a kind of function. The programmer can then specify which components (or compositions of components) must execute atomically, and our tool will do the rest. Right now, programmers have to specify locking manually. We plan to integrate LOCKPICK with Flux in the near future.

5. Conclusion

We have presented a system for inferring locks to support atomic sections in concurrent programs. Our approach uses points-to and effects analysis to infer those locations that are shared between threads. We then use mutex inference to determine an appropriate set of locks for protecting accesses to shared data within an atomic section. We have proven that mutex inference provides the same amount of parallelism as if we had one lock per location.

In addition to the aforementioned ideas for making our approach more efficient, it would be interesting to understand how optimistic and pessimistic concurrency controls could be combined. In particular, the former is much better and handling deadlock, while the latter seems to perform better in many cases [9]. Using our algorithm could help reduce the overhead and limitations (e.g., handling I/O) of an optimistic scheme while retaining its liveness benefits.

References

[1] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: A Language for Programming High-Performance Servers. In *In Proceedings of the Usenix Annual Technical Conference*, 2006. To appear.

[2] C. Flanagan and S. N. Freund. Automatic synchronization correction. In *Synchronization and Concurrency in Object- Oriented Languages (SCOOL)*, Oct. 2005.

[3] C. Flanagan, S. N. Freund, and M. Lifshin. Type Inference for Atomicity. In *TLDI*, 2005.

[4] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *PLDI*, 2003.

[5] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA ‘O3*, pages 388–402, Oct. 2003.

[6] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP ‘05*, June 2005.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *PODC ‘03*, pages 92–101, July 2003.

[8] J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *POPL*, 1988.

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

[10] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *Proceedings of the 2006 PLDI*, Ottawa, Canada, June 2006. To appear.

[11] J. Rehof and M. Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *POPL*, 2001.

[12] M. F. Ringenburg and D. Grossman. Atomcaml: First-class atomicity via rollback. In *ICFP ‘05*, pages 92–104, Sept. 2005.

[13] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *ECOOP ‘O4*, Oslo, Norway, 2004.