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

Software transactions allow the programmer to specify sections of code that should be serializable, without the programmer needing to worry about exactly how atomicity is enforced. Recent research proposes using optimistic concurrency to implement transactions. In this short paper, we propose a pessimistic lock-based technique that uses the results of static whole-program analysis to enforce atomicity. The input to our analysis is a program that contains programmer-specified atomic sections and calls to fork. We present a sharing inference algorithm that uses the results of points-to analysis to determine which memory locations are shared. Our analysis uses *continuation effects* to track the locations accessed after a point in the program. This allows data to be thread-local before a fork and thread-shared afterward. We then present a mutex inference algorithm that determines a sufficient set of locks to guard accesses to shared locations. After mutex inference, a compiler adds the appropriate lock acquires and releases to the beginning and end of atomic sections. Our algorithm is efficient, and provides parallelism according to precision of the alias analysis while minimizing the number of required locks.

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 parallelism and at worst in deadlock, 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, 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 atomic with respect to the other atomic sections 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 frees programmers from many low-level concerns.

Recent research proposes to implement atomic sections using optimistic concurrency techniques [5, 7, 13, 6, 12]. Roughly speaking, memory accesses within a transaction are logged, and the log must be consistent with the current state of memory at the conclusion of the transaction; if not, the transaction is rolled back and restarted. The main drawback with this approach are that it does not interact well with I/O, which cannot always be rolled back. While performance can be quite good [12], optimistic concurrency performance can also be quite a bit slower than traditional techniques due to the costs of logging and rollback [9].

In this short paper, we explore using pessimistic techniques based on the results of a static program analysis to implement atomic sections. We assume 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 statements in s to enforce atomicity while avoiding deadlock. An important goal of our algorithm is to maximize parallelism. A trivial implementation would be to begin and end all atomic sections by, respectively, acquiring and releasing a single global lock. We present an improved algorithm that uses much finer locking but still enforces atomicity. 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. Suppose we perform a points-to analysis on the program. This 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 alias 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 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 [10] as a post-pass.

The algorithm we present here performs two optimizations over the basic approach sketched above. 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 [8] analysis to determine the shared abstract locations as well as the locations accessed

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

Figure 1. Source Language, Types, and Constraints

within an atomic section (Section 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. This sharing inference is also used by Locksmith, a race detection tool for C programs [10]. The sharing analysis presented here is essentially unchanged from Locksmith's sharing analysis, 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 potentially uses many fewer locks.

The remainder of the paper describes our lock inference algorithm in detail.

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 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 two kinds of labels: *location labels* ρ and *effects* ε and χ, which represent those locations ρ dereferenced or assigned to during a computation. Typing a program generates *label flow constraints* $l \leq l'$ and afterward these constraints are solved to learn the desired information. The constraint $l \leq l'$ is read "label l flows to label l'." For example if `x` has type $\mathit{ref}^{\,\rho}\ \tau$, and we have constraints $\rho' \leq \rho$ and $\rho'' \leq \rho$, then `x` may point to the locations ρ' or 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 simple type τ annotated with the effect χ, which approximates the effect of evaluating e at run time. Within the type rules, the judgment $C \vdash l \leq l'$ indicates that $l \leq l'$ can be proven by the constraint set C. In an implementation, such judgments cause us to "generate" constraint $l \leq l'$ and add it C. Because we assume a call-by-value semantics, all of the assumptions in type environment Γ refer to values, and thus are given simple types. Simple types include standard integer types; updatable reference types $\mathit{ref}^{\,\rho}\ \tau$, which is decorated with a location label ρ; and function types of the form $(\tau, \varepsilon) \rightarrow^{\chi} (\tau', \varepsilon')$, 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 $\texttt{fork}\ e$ 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).

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:

$$
\begin{array}{l}
\texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in} \\
\texttt{let}\ y = \texttt{ref}\ 1\ \texttt{in} \\
\quad x := 4; \\
\quad \texttt{fork}^1\ (!\,x; !\,y); \\
\quad /*\,(1)\,*/ \\
\quad y := 5
\end{array}
$$

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 to read x and y. The effect of the output effect of the fork (i.e. starting at (1)) is to write y. Thus 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.

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], we pick some labels ε_{in} and ε_{out} for the input and output effects of the function, and bind them in the type. 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 evalua-

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

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

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

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

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

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

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

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

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

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

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

Figure 2. Type Inference Rules

tion: 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. In an actual implementation we always pick location label ρ to be a fresh label. [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 $C \vdash \rho \leq \varepsilon'$. In addition we require $\rho \leq \chi$. Note that [Sub] can be applied before

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

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

$$\text{[Sub-Fun]}\ \frac{\begin{array}{c}C \vdash \tau_2 \leq \tau_1 \qquad C \vdash \tau'_1 \leq \tau'_2 \\ C \vdash \varepsilon_1 \leq \varepsilon_2 \quad C \vdash \varepsilon'_2 \leq \varepsilon'_1 \quad C \vdash \chi_1 \leq \chi_2\end{array}}{C \vdash (\tau_1,\varepsilon_1) \rightarrow^{\chi_1} (\tau'_1,\varepsilon'_1) \leq (\tau_2,\varepsilon_2) \rightarrow^{\chi_2} (\tau'_2,\varepsilon'_2)}$$

Figure 3. Subtyping Rules

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.

[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 $\chi_1 \leq \chi$ and $\chi_2 \leq \chi$ 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 $C \vdash \tau \leq \tau'$, 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^i_e 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, we will compute $\varepsilon^i_e \cap \varepsilon^i$; 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^i_e 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
  fork^1 (!x);
  /*(1)*/
  fork^2 (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 in-

ference. We believe it is straightforward to show that continuation effects are a sound approximation of the locations accessed by an expression.

2.2 Computing Sharing

After applying the type inference rules in Figures 2 and 3, we are left with a set of label flow constraints C. We can think of these constraints as forming a directed graph, where each label forms a node and constraints $l \leq l'$ is represented as a directed edge from l to l'. Then for each label l, we can compute the set $S(l)$ of location labels ρ that "flow" to l by transitively closing the graph. This can be done by performing one depth-first search for each node in the graph. The total time 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.)

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

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

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 of our algorithm is to compute a set of locks to use to guard all of the shared locations. A simple and correct solution is to associate a lock ℓ_ρ with each shared location $\rho \in \textit{shared}$. 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 naive approach of using a single lock for all atomic sections.[1] 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. To improve this basic approach while retaining the same level of parallelism, we can exploit 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 simply acquire ρ's mutex in an atomic section, since doing so will implicitly protect ρ' as well. 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 $O(m|\textit{shared}|)$, where m is the number of atomic sections. Thus in total this takes time $O(m|\textit{shared}|^2)$ for all locations.

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

Given the dominates relationship, we can 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.*

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

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 gives us an optimal result, no matter the ordering. Mutex selection takes time $O(|\textit{shared}|^2)$, 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 $\textit{range}(L)$. 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 $O(m|\textit{shared}|^2)$ 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 $L(c) = \ell_b$. Then eventually we will 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 naive 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 and Optimality

It should be clear that the algorithm for computing the dominates relationship is correct. Recall that the goal of mutex selection is to

ensure that no shared location can be accessed without at least one lock consistently held. We can define this formally as follows, with respect to the alias analysis. 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*

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

In other words, the parallelism of a program is all possible pairs of atomic sections that could execute completely in parallel because they access no common locations. We define $L(S_i) = \{L(\rho) \mid \rho \in S_i\}$.

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

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

In words, $P(L)$ is 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.

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 $L_0(\rho) = \ell_\rho$. 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 $\rho_2 \geq \rho_1$.

□

LEMMA 2 (Correctness and Optimality). *We claim $P(L) = P$. 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 $P = P(L_k)$ 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.

(Correctness) Suppose $S_i \cap S_j \neq \emptyset$. Then clearly there is a $\rho'' \in S_i \cap S_j$, and so trivially $L_{k+1}(S_i) \cap L_{k+1}(S_j) \neq \emptyset$.

(Optimality) Otherwise suppose $S_i \cap S_j = \emptyset$. Then $L_k(S_i) \cap L_k(S_j) = \emptyset$, 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 $L_{k+1}(S_i) \cap L_{k+1}(S_j) = \emptyset$. 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 since $\rho' > \rho$, we have $\rho' \in S_i$. But 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 $\rho'' \in S_j$ such that $L_k(\rho'') = \ell_{\rho'}$. But then by Lemma 1, we have $\rho' \geq \rho''$. But then $\rho' \in S_j$, a contradiction. Hence we must have $\ell_{\rho'} \notin L_k(S_j)$, and therefore $L_{k+1}(S_i) \cap L_{k+1}(S_j) = \emptyset$. □

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. One choice 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.

We are currently building an implementation of our algorithm as part of Locksmith. Our approach is 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 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.

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. 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.

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: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, New York, NY, USA, 2006. ACM Press.

[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.