Dynamically Checking Ownership Policies in Concurrent C/C++ Programs

Jean-Philippe Martin
Microsoft Research
jpmartin@microsoft.com

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

Manuel Costa
Microsoft Research
manuelc@microsoft.com

Periklis Akritidis
University of Cambridge
pa280@cl.cam.ac.uk

Miguel Castro
Microsoft Research
mcastro@microsoft.com

Abstract

Concurrent programming errors arise when threads share data incorrectly. Programmers often avoid these errors by using synchronization to enforce a simple ownership policy: data is either *owned exclusively* by a thread that can read or write the data, or it is *read owned* by a set of threads that can read but not write the data. Unfortunately, incorrect synchronization often fails to enforce these policies and memory errors in languages like C and C++ can violate these policies even when synchronization is correct.

In this paper, we present a dynamic analysis for checking ownership policies in concurrent C and C++ programs despite memory errors. The analysis can be used to find errors in commodity multi-threaded programs and to prevent attacks that exploit these errors. We require programmers to write ownership assertions that describe the sharing policies used by different parts of the program. These policies may change over time, as may the policies' means of enforcement, whether it be locks, barriers, thread joins, etc. Our compiler inserts checks in the program that signal an error if these policies are violated at runtime. We evaluated our tool on several benchmark programs. The run-time overhead was reasonable: between 0 and 49% with an average of 26%. We also found the tool easy to use: the total number of ownership assertions is small, and the asserted specification and implementation can be debugged together by running the instrumented program and addressing the errors that arise. Our approach enjoys a pleasing modular soundness property: if a thread executes a sequence of statements on variables it owns, the statements are serializable within a valid execution, and thus their effects can be reasoned about in isolation from other threads in the program.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Testing and Debugging—Testing tools; F.3.2 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification Techniques

General Terms Reliability, Security, Theory, Verification.

Keywords Concurrency, Debugging, Dynamic analysis, Security, Testing, Tools

1. Introduction

Concurrent programming errors arise when the executions of two or more threads interfere while operating on the same data. Programmers avoid these problems by using synchronization mechanisms to enforce a simple policy: a thread may access a piece of data so long as it *owns* it. A single thread can own memory *exclusively*, meaning the thread may read and write the data, or many threads can own the data *collectively*, meaning that all may read it, but none may write it. Unfortunately, incorrect synchronization often fails to enforce these ownership policies and memory errors in languages like C and C++ can violate these policies even when synchronization is correct.

This paper develops a dynamic analysis for checking that memory ownership policies are properly enforced in multi-threaded C and C++ programs despite memory errors. The analysis can be used to find unwanted interference during concurrent executions and to prevent attacks that exploit these errors. To use our analysis, a programmer adds assertions to indicate the current ownership policy for a piece of data. For example, suppose the program uses a shared queue that contains jobs to be processed by worker threads. We might annotate the worker thread code with assertions as

```
1  struct job *j = dequeue(q);
2  ownEx(j, sizeof(*j));
3  /* ... process job, accessing *j freely ... */
4  relEx(j, sizeof(*j));
5  enqueue(q,j);
```

After extracting the job from the queue, on line 2 the current thread asserts that it owns j's data exclusively—no other thread should access j's data at this point. After processing the job, the thread releases ownership (line 4) and places the job back in the queue. Another thread may now dequeue the job and take ownership of it for further processing.

Ownership assertions constitute a *specification*. It is up to the programmer to implement this specification. For example, to implement the specification above, the programmer must ensure that the queue does not have more than one pointer to the same job and

* Work performed while this author was visiting Microsoft Research, Cambridge, and the University of Cambridge Computer Laboratory.

POPL'10, January 17–23, 2009, Madrid, Spain.

dequeue must use synchronization to ensure that only one thread can extract a particular job. Our dynamic analysis checks that the programmer implements the ownership specification. In particular, given a program containing ownership assertions, our run-time system tracks the ownership state of each memory location, and our compiler instruments reads and writes to check whether they respect the target data's ownership state. For the example above, the ownEx assertion would change the ownership state of j's data to grant exclusive ownership to thread t. Thus, the inserted checks would signal an ownership violation if another thread attempted to access j's data, e.g., as the result of an error in the queue implementation or a memory error anywhere in the program. We also insert checks to ensure that ownership assertions are themselves legal, e.g., it is a violation for one thread to claim exclusive ownership of a location already owned by another thread.

We formalize the semantics of ownership checking on multi-threaded program traces (Section 3), and prove that ownership assertions can be used to establish *serializability*: if a thread t successfully acquires ownership of some set of memory locations L, then we can prove that subsequent operations on L will appear as if they executed serially, without interleaving by (valid) operations of other threads, until ownership is released. Our serializability property is modular in the sense that it holds regardless of what other threads do: a properly annotated block will either exhibit a serial execution or the program will fail due to an ownership violation by another thread.

Previous work proposes tools to check that code properly uses synchronization to implement atomic blocks or lower-level data sharing patterns. SharC [2] and its successor Shoal [3] specify sharing policies by annotating types in C/C++ programs, using dynamic casts to change policies. These policies are checked using a combination of static and dynamic analysis. They provide weaker guarantees than our analysis because they fail to detect policy violations due to memory errors. Additionally, they tie ownership policies to specific synchronization constructs and impose restrictions on dynamic policy changes that can be cumbersome. Our annotations are conceptually simpler and more expressive. The number of annotations required by our analysis and its performance overhead are comparable to those in Shoal. There are also atomicity checking tools such as Velodrome [11] and Atomizer [9] for type safe languages like Java. They provide strong guarantees but can slow down execution by several factors.

We have implemented our analysis as a compiler and run-time system for multi-threaded C and C++ programs (Section 4), extending our work on preventing memory errors [1]. We evaluated our tool on seven benchmark programs, totaling more than 265 KLOC (Section 5). Our experiments reveal four benefits of our system:

Expressiveness. We found that ownership assertions were expressive enough to capture sharing patterns used in practice. Notably, our benchmarks often employed dynamic changes in ownership, such as the ownership transfer between threads via the shared queue in our example.

Ease of use. Annotating programs was never burdensome; in the worst case, the added assertions and small program changes were 6% of the program size, and they were easy to come by. In total, we annotated or changed roughly 500 lines in all benchmarks, constituting less than 0.2% of the total program sizes. By default, data is exclusively owned by the thread that allocated it, so executing a program without assertions quickly reveals sharing patterns that we can convert into reasonable specifications.

Low overhead. The overhead of our dynamic analysis is reasonable: programs are slower by 26%, and use 10% more memory, on average, compared to the original versions. We achieve low overheads by using efficient datastructures to perform ownership checking, by avoiding atomic operations in most ownership checks, and by eliminating redundant ownership checks at compile-time. We prove that these optimizations preserve our serializability result.

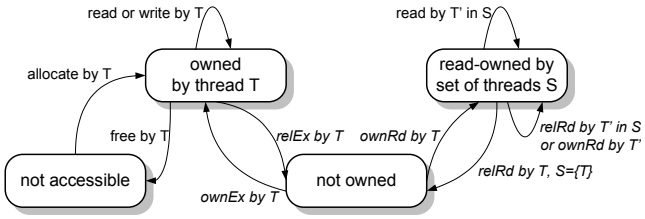

Figure 1. Core ownership states.

Utility. In testing our benchmark programs, we found several errors, both memory errors and concurrency errors. Our overhead is low enough for some programs that we could imagine deploying these programs with ownership checking enabled, towards improving both software security and our ability to diagnose concurrency errors in the field.

2. Dynamic Ownership Checking

This section overviews our dynamic ownership analysis. We describe basic ownership policies, some extensions to shorten specifications and improve performance, and target applications.

Basic ownership policies. Memory ownership policies place each byte of memory into one of four states, which determine the legal operations on that memory, as depicted in Figure 1: *inaccessible*; *owned* exclusively by a single thread permitted to read and write it; *read-owned* by possibly several threads permitted to read it; and *not owned* by any thread. Memory begins as inaccessible, and then is exclusively owned by the thread that allocates it. Exclusive ownership is explicitly released by the relEx assertion, which places memory in the unowned state. From this state, a thread could either claim exclusive ownership again via ownEx, or could claim read-ownership via ownRd. From the read-owned state, other threads may assert read-ownership of the same data. Read ownership is released through relRd; when all read-owners have released ownership, the memory returns to the unowned state. Accesses to unowned memory are illegal.

Figure 2 shows ownership-annotated code for a FIFO queue based on similar code in the `pfscan` benchmark. Ignoring the ownership assertions for the moment, we can see the basic code implements a standard thread-shared queue. The Q structure stores queued pointers in a fixed-size array accessible from its buf field, and synchronizes access to the array using a lock stored in the mtx field. The current and maximum size of the queue are stored in the occupied and size fields, respectively, and nextin records the next free slot in buf. When enqueuing an item, the enqueue code first checks whether the queue is full (line 23); if so, it waits on condition variable q→less to be signaled by dequeue when an item is removed. The code then stores the item (line 29), performs some bookkeeping (lines 31–32), and on the last line signals condition variable q→more to awaken any threads waiting in dequeue for items to become available.

The ownership assertions together specify the high-level policy that only one thread at a time may access the queue. For example, at line 22 the assertion ownEx(q,DATA_SZ) specifies that at this point, the current thread should have exclusive ownership over the first four fields of the q structure. Line 28 similarly asserts the current thread exclusively owns q→buf. The programmer enforces this policy by using q→mtx to mediate access to this data: as long as all accesses take place with q→mtx held, only one thread will ever access the data at a time. Notice that ownership is released prior

```
1  typedef struct Q {
2      int size, occupied, nextin;
3      void ** buf;
4  #define DATA_SZ offsetof(Q,mtx)
5      Mutex mtx;
6      ConditionVariable less, more;
7  #define SYNC_SZ sizeof(Q)−DATA_SZ
8  } Q;
9
10 Q* queueInit(size_t n) {
11     Q* ret = malloc(sizeof(Q));
12     ret→ buf = malloc(n*sizeof(void*));
13     initQheader(ret, n);
14     relEx(ret, DATA_SZ);
15     makeRO(&ret→mtx, SYNC_SZ);
16     relEx(ret→buf, n*sizeof(void*));
17     return ret;
18 }
19
20 void enqueue(Q *q, void *item) {
21     lock(&q→ mtx);
22     ownEx(q, DATA_SZ);
23     while (q→ occupied ≥ q→ size) {
24       relEx(q, DATA_SZ);
25       condwait(&q→ less, &q→ mtx);
26       ownEx(q, DATA_SZ);
27     }
28     ownEx(&q→buf[q→nextin], sizeof(void*));
29     q→ buf[q→ nextin] = item;
30     relEx(&q→buf[q→nextin], sizeof(void*));
31     q→ nextin = (q→ nextin+1) % q→ size;
32     q→ occupied++;
33     relEx(q, DATA_SZ);
34     unlock(&q→ mtx);
35     signal(&q→ more);
36 }
37
38 Q *q;
39 main() {
40     q = queueInit(20);
41     makeRO(&q, sizeof(q)); ...
42 }
```

Figure 2. Code sample.

to releasing the lock via calls to condwait (line 24), and unlock (line 33), and acquired just after acquiring the lock (line 22) or reacquiring it via condwait (line 26). The objects put in the queue are protected using ownership assertions in client code—as shown in the Introduction, the client releases ownership before enqueuing a job, and asserts ownership when dequeuing one.

Checking correct enforcement. We would like to check whether a program properly enforces its declared ownership policy. In other words, is it true that under all possible executions, when some thread t is executing lines 23, 29, 31, or 32, no thread but t will access q's first four fields or its q→ buf array? The dynamic nature of ownership policies and C's lack of memory safety would make it difficult to write a static checker that is sound (no missed errors), precise (few false alarms), and scalable (works to large programs). Therefore, we choose to check proper ownership policy enforcement during program execution. Doing so allows our tool to be precise and scalable, though potentially missing some latent errors. In our approach, we associate shadow state with the program's run-time memory, and implement ownership assertions to change that state according to Figure 1. We compile the program to insert checks to verify that reads, writes, and ownership transitions are legal according to the target memory's ownership state, as shown in the figure.

```
1  Q* queueInit(size_t n) {
2      Q* ret = malloc(sizeof(Q));
3      Cluster cluster = allocCluster();
4      ret→ buf = malloc(n*sizeof(void*));
5      initQheader(ret,n);
6      giveToCluster(ret, DATA_SZ, cluster);
7      makeRO(&ret→mtx, SYNC_SZ);
8      giveToCluster(ret→buf, n*sizeof(void*),cluster);
9      return ret;
10 }
11
12 #define lockCluster(mtx,ptr) lock(mtx);ownClusterEx(ptr)
13 #define unlockCluster(mtx,ptr) relClusterEx(ptr);unlock(mtx)
14 #define condwaitCluster(cond,mtx,ptr) \
15       relClusterEx(ptr); condwait(cond,mtx); ownClusterEx(ptr)
16
17 void enqueue(Q *q, void *item) {
18     lockCluster(&q→mtx,q);
19     while (q→ occupied ≥ q→ size) {
20         condwaitCluster(&q→less, &q→mtx, q);
21     } ...
22     unlockCluster(&q→mtx,q);  ...
23 }
```

Figure 3. Cluster code fragment.

Even though the FIFO code snippet reveals no bug, our dynamic checker can reveal problems in other parts of the code that would violate the intended policy. For example, some other thread might access *q without first grabbing the queue mutex q→ mtx. More perniciously, an out-of-bounds access anywhere in the code might cause some other thread to overwrite q or what it points to. In both cases, our inserted checks would signal violations of the declared policy; we halt execution at the point of violation, making it easy to find the source of the problem.

As hinted by these examples, and proved in the next section, it turns out that the core of the enqueue function enjoys the following pleasingly modular property: either the execution of lines 26–33 will be *serializable*—in that their effects will be just as if they were executed serially, despite interleaving with other threads—or there will be an ownership violation, e.g., because some other thread illegally modified the q data structure. A key contribution of our paper is to show how to provide this property regardless of what other threads do. As we detail in Section 4, we implement several mechanisms to prevent threads from bypassing our checks, corrupting heap management, or corrupting the shadow state.

A nice feature of our system is that ownership assertions are easy to add or debug by simply running the code and addressing the resulting violations. For example, we might have thought that enqueue should acquire the *entire* queue structure exclusively, and not just its first four fields. But on running the code under such a policy, we would discover an ownership violation when a second thread tried to synchronize on the q→ mtx field and/or the condition variables. Hence we annotate queueInit to assert these fields are *read-only for all threads* by adding makeRO(&ret→ mtx,SYNC_SZ) on line 15. This notational shorthand allows each thread to avoid explicitly acquiring read-only ownership of the mutex each time it goes to synchronize on it. We would likewise discover that the global variable q needs to be read-only, e.g., to access mtx, and so assert it is read-only on line 41. Finally, we would find that some data in the mutex structure pointed to by q→ mtx may be read/written by several threads at once, using operations like compare-and-swap to implement the lock's semantics. Therefore we place this data in a special *unchecked* state (not shown in Figure 1), permitting any thread to read or write it. This state is also useful for program variables subject to *benign races*, e.g., performance counters.

```
1  void HandleSession(int socket) {
2      int uid = Authenticate(socket);
3      UserInfo* uinfo = &userTable[uid]
4      ProcessRequests(uinfo, socket);
5  }
6
7  ProcessRequests(UserInfo* uinfo, int socket) {
8      // loop processing requests
9      while (true) switch (command) {
10         ...
11         case buy: ...
12         case deleteAccount:
13             // allow uinfo to be reused
14             memset(uinfo, 0, sizeof(UserInfo));
15             return;
16         ...
17     }
18 }
```

Figure 4. An exploitable concurrency error.

Clusters. A single ownership assertion like ownEx applies to a contiguous chunk of memory. As a result, several assertions would be needed to take/release ownership of entire linked data structures, which is tedious to the programmer and adds run-time overhead. To alleviate these concerns, we extend our system with the abstraction of a *cluster* of memory chunks. Threads can dynamically allocate clusters and assign chunks of memory they exclusively own to a cluster using the assertion giveToCluster. Having done this, a thread can release or own the entire cluster at once using assertions relClusterEx and ownClusterEx, respectively. Each call takes a pointer as its argument and changes the state of the cluster of the pointed-to memory. A memory location can be in at most one cluster at any one time. (The full state graph, which includes clusters and unchecked states, is shown later in Figure 9.)

Figure 3 shows what some of the queue code would look like using clusters and macros. The queueInit function allocates a cluster for the queue (line 3), then assigns the queue's relevant memory to that cluster (lines 6, 8). The macros (lines 12–14) simplify annotating locks; similar macros can be written for other synchronization primitives. With clusters and macros, the enqueue function is again very simple—it now has three total assertions that are comingled with the three synchronization calls (lines 18, 20, 22), and does not need separate assertions to take ownership of buf since it belongs to the cluster. Although the example does not show it, it is possible to take read-ownership of clusters.

Our experience is that code does not require many ownership assertions. Figure 2 is atypical in this respect: we chose it specifically because it shows many assertion types. The worst-case benchmark has fewer than 4% of annotation lines, and on average we annotate only 0.1% of the lines.

Applications. We envisage several applications for our dynamic ownership analysis. The most obvious is finding concurrency and memory errors during development and testing. Our analysis can also be used during production to diagnose errors in the field [13] and to improve security. Since our dynamic analysis ensures control-flow integrity and detects sequential buffer overflows and underflows [1], it prevents the most common attacks that exploit low level defects in C and C++ programs. Our analysis can also prevent some exploits of concurrency errors. For example, consider the code in Figure 4, which illustrates a hypothetical session-based commerce server. Each session is handled by a different thread but the programmer assumed incorrectly that each user has at most one active session. Suppose a user A somehow starts two sessions, each pointing to the same slot in userTable, and one of the sessions deletes the account. If a new user B reuses

Domains

$u, t \in Tid \quad x \in Var \quad v \in Value \quad e \in Expr$
$\sigma \in Heap = Var \rightarrow Value$

Operations

$$\begin{array}{lll} a \in Operation & ::= & \mathsf{rd}(t,x,v) \mid \mathsf{wr}(t,x,v) \\ & \mid & \mathsf{sync}(t, op, x, v_1 ... v_n) \mid b \\ b \in OwnOper & ::= & ... \\ \alpha \in Trace & ::= & \cdot \mid a, \alpha \end{array}$$

Semantics

$$\begin{array}{lll} \text{(Read)} & \sigma \longrightarrow^{\mathsf{rd}(t,x,v)} & \sigma \quad \text{where } \sigma(x) = v \\ \text{(Write)} & \sigma \longrightarrow^{\mathsf{wr}(t,x,v)} & \sigma[x \mapsto v] \\ \text{(Sync)} & \sigma \longrightarrow^{\mathsf{sync}(t,op,x,v_1 ... v_n)} & \delta_{op,x,t}(\sigma, v_1 ... v_n) \\ & & \text{where } \phi_{op,x,t}(\sigma, v_1 ... v_n) \\ \text{(Monitor)} & \sigma \longrightarrow^{b} & \sigma \end{array}$$

$$\text{[TSTEP]} \quad \frac{a \vdash e_i \longrightarrow e_i' \qquad \sigma \longrightarrow^{a} \sigma' \qquad tid(a) = i}{\sigma; e_1 \| ... e_i ... \| e_n \longrightarrow^{a} \sigma'; e_1 \| ... e_i' ... \| e_n}$$

Figure 5. Semantics of Multithreaded Programs

that slot in the userTable, user A could use its second session to buy goods using B's account. Our tool forces the programmer to make all assumptions about sharing explicit. In the example, the programmer would have to assert that uinfo is owned exclusively to allow reads and writes during request processing. Therefore, our tool would prevent user A from exploiting this error by signalling a policy violation when the user starts the second session.

3. Formalism

This section presents a formalism to describe our dynamic ownership checking analysis. The presentation proceeds in three steps. First, we define a semantics of multi-threaded traces, where a trace is a sequence of basic operations (e.g., reads, writes, and synchronization operations) performed by any number of threads, and characterizes their effect on memory.

Second, we define ownership checking as a judgment on traces, referring to this judgment as *trace validation*. In our implementation, we perform ownership checking at run-time, essentially comingling trace validation with the operational semantics. Formalizing these two separately simplifies the theoretical development. We prove that in valid traces, sequences of operations by a single thread on locations owned by that thread can be serialized.

Finally, we present an alternative *relaxed* semantics in which ownership checks are made first-class operations that appear in traces, allowing them to be implemented more efficiently, and potentially optimized away. In the basic semantics, trace validation views a read (or other operation) as occurring atomically along with the necessary ownership check. Making the ownership check a separate operation avoids forcing the check and the operation to be performed atomically. We prove that when separating the check from all but the operations for acquiring ownership, the system is sound and complete with respect to the basic semantics, so that once again validity implies serializability. We also prove that many ownership checks can be removed without compromising trace validity.

3.1 Multithreaded program traces

The semantics of multithreaded programs is shown in Figure 5. It defines legal operations a on memory σ. We make no distinction between global and local (per-thread) memory, nor do we consider

allocating/deallocating memory. Memory is indexed by variables x, which can be viewed as memory addresses. An operation is either a read or write of a value v by a thread t from/to a variable x, a synchronization operation op by a thread t that may read or write variable x, or an ownership-related operation b, the details of which we consider in the next subsection.

The semantics defines two judgments. The judgment $\sigma \longrightarrow^a \sigma'$ defines the effect of operation a on a memory σ. Reads and writes on variables have the obvious semantics. The semantics of synchronization operations is parametrized by a predicate ϕ and a function δ, indexed by the operation's details. In particular, thread t may perform synchronization operation op involving variable x so long as the heap satisfies the predicate $\phi_{op,x,t}$; as a result the heap is transformed according to the function $\delta_{op,x,t}$. If the operation requires additional parameters, they are passed to the function/predicate as needed. As an example, we could model a semaphore x with operations *seminit*, *wait* and *signal*, defining δ and ϕ as follows:

$$\begin{array}{lcl}
\phi_{seminit,x,t} & \equiv & \lambda(\sigma, n).\ \mathit{true} \\
\delta_{seminit,x,t} & \equiv & \lambda(\sigma, n).\ \sigma[x \mapsto n] \\
\phi_{wait,x,t} & \equiv & \lambda\sigma.\ \sigma(x) > 0 \\
\delta_{wait,x,t} & \equiv & \lambda\sigma.\ \sigma[x \mapsto \sigma(x) - 1] \\
\phi_{signal,x,t} & \equiv & \lambda\sigma.\ \mathit{true} \\
\delta_{signal,x,t} & \equiv & \lambda\sigma.\ \sigma[x \mapsto \sigma(x) + 1]
\end{array}$$

As one can use semaphores to implement mutual exclusion locks, reader/writer locks, barriers, and other synchronization operations, it should be evident that using δ and ϕ to model synchronization operations is suitably expressive for our purposes.

The second judgment $\sigma; e_1 \| ... e_i ... \| e_n \longrightarrow^a \sigma'; e_1 \| ... e_i' ... \| e_n$ defines the execution of a multi-threaded program where a (nondeterministically chosen) thread e_i takes a step, potentially modifying the heap. We do not define individual thread transitions, but write $a \vdash e \longrightarrow e'$ to indicate that e transitions to e', according to the action a.

A trace α is a list of actions a_i, and is termed an *execution* if it arises from the semantics.

Definition 1 (Execution)**.** *A trace $\alpha = a, a', ..., a''$ is an* execution *iff there exists some $\sigma, \sigma', ..., \sigma''$ such that $\sigma \longrightarrow^a \sigma' \longrightarrow^{a'} ... \longrightarrow^{a''} \sigma''$.*

3.2 Basic trace validation

The formalism defines basic operations b for owning memory exclusively or for reading-only. Additionally modeling clusters and permanent read-only states would be straightforward.

Ownership checking is specified as a judgment on traces, $\omega, \rho \ \vdash_0 \ \alpha \rightsquigarrow \omega', \rho'$, which we call *trace validation*, shown in Figure 6. (As mentioned earlier, in practice we interleave ownership checking with actual execution.) Here, ω and ω' are *write-ownership maps* (or, simply *write maps*) from variables x to the thread that owns them for writing. ρ and ρ' are *read-ownership maps* (or, simply *read maps*) from variables to sets of threads that are allowed to read the variable. If no thread exclusively owns a variable x, we have $\omega(x) = \bot$. Only variables owned by $\top$ may be used in synchronization operations. Ownership by $\top$ models the *unchecked* state described in the previous section, needed because synchronization operations often fail to obey proper ownership. Ownership by $\top$ is invariant throughout a program execution. We write $\omega_\bot^S$ for a write map that maps all variables to unowned except those in S, which are used for synchronization; we write $\omega_\bot$ when the precise definition of S is unimportant. We use $\rho_\emptyset$ to indicate the empty read map, which maps all variables to the empty-set $\emptyset$.

In the judgment, ω and ρ represent the maps prior to validating trace α, and ω' and ρ' indicate the state of the maps at the conclusion of validation. Rule [C-Trace] appeals to the judgment

[C-OwnEx]
$$\frac{\omega(x) = \bot \qquad \rho(x) = \emptyset}{\omega, \rho \ \vdash_0 \ \mathsf{ownEx}(t, x) \rightsquigarrow \omega[x \mapsto t], \rho[x \mapsto \{t\}]}$$

[C-OwnRd]
$$\frac{\omega(x) = \bot}{\omega, \rho \ \vdash_0 \ \mathsf{ownRd}(t, x) \rightsquigarrow \omega, \rho[x \mapsto \rho(x) \cup \{t\}]}$$

[C-RelEx]
$$\frac{\omega(x) = t}{\omega, \rho \ \vdash_0 \ \mathsf{relEx}(t, x) \rightsquigarrow \omega[x \mapsto \bot], \rho[x \mapsto \emptyset]}$$

[C-RelRd]
$$\frac{t \in \rho(x)}{\omega, \rho \ \vdash_0 \ \mathsf{relRd}(t, x) \rightsquigarrow \omega, \rho[x \mapsto (\rho(x) \setminus \{t\})]}$$

[C-Read]
$$\frac{t \in \rho(x)}{\omega, \rho \ \vdash_0 \ \mathsf{rd}(t, x, v) \rightsquigarrow \omega, \rho}$$

[C-Write]
$$\frac{\omega(x) = t}{\omega, \rho \ \vdash_0 \ \mathsf{wr}(t, x, v) \rightsquigarrow \omega, \rho}$$

[C-Sync]
$$\frac{\omega(x) = \top}{\omega, \rho \ \vdash_0 \ \mathsf{sync}(t, op, x, v_1 ... v_n) \rightsquigarrow \omega, \rho}$$

[C-Trace]
$$\frac{\omega, \rho \ \vdash_0 \ a \rightsquigarrow \omega', \rho' \qquad \omega', \rho' \ \vdash_0 \ \alpha \rightsquigarrow \omega'', \rho''}{\omega, \rho \ \vdash_0 \ a, \alpha \rightsquigarrow \omega'', \rho''}$$

where

$$\begin{array}{rcl}
b \in \mathit{OwnOper} & ::= & \mathsf{ownEx}(t, x) \mid \mathsf{ownRd}(t, x) \\
 & \mid & \mathsf{relEx}(t, x) \mid \mathsf{relRd}(t, x) \\
\omega \in \mathit{ExclOwnership} & = & \mathit{Var} \to (\mathit{Tid} \cup \{\top, \bot\}) \\
\rho \in \mathit{ReadOwnership} & = & \mathit{Var} \to 2^{\mathit{Tid}} \\
\omega_\bot^S & = & \lambda x.\mathsf{if}\ x \in S\ \mathsf{then}\ \top\ \mathsf{else}\ \bot \\
\rho_\emptyset & = & \lambda x.\emptyset
\end{array}$$

Figure 6. Basic trace validation

$\omega, \rho \ \vdash_0 \ a \rightsquigarrow \omega', \rho'$ to validate an individual action a, where the maps resulting from validating a are then used to validate the remainder of the trace α. The rules for validating operations are straightforward, following the transition diagram in Figure 1. Notice that rule [C-OwnEx] checks/modifies both maps ω and ρ, since exclusive ownership grants both read and write access. One might expect $\rho(x) = \{t\}$ as an additional premise of [C-RelEx], and $\omega(x) = \bot$ as a premise of [C-RelRd], symmetrical to the modifications to the maps made by [C-OwnEx] and [C-OwnRd], respectively. Without these checks, a legal trace may perform an $\mathsf{ownEx}(t, x)$ followed by $\mathsf{relRd}(t, x)$, so that x may only be accessed for writing from then on. While perhaps odd, removing the checks simplifies the development of relaxed checking in the next subsection, and poses no problems for soundness.

Serializability. The main property we prove is *serializability*. In particular, by using ownership specifications in a particular way, a programmer can ensure that, for valid traces, a sequence of thread operations will always execute in a manner that is equivalent to one in which the operations occur in sequence. More precisely, a code sequence will be serializable if it adheres to a discipline of *two-phase ownership*, in which all ownEx (and ownRd) operations strictly precede relEx (and relRd) operations, with reads/writes on owned data interleaved among them. For example, the code snippet

in the introduction trivially follows this discipline, as there is a single ownEx followed by some reads/writes and finally a single relEx. Lines 26–33 in Figure 2 exhibit two-phase ownership as well, this time starting with two ownEx operations, with reads/writes interleaved between the two final relEx operations.

We prove serializability using the method of reduction due to Lipton [16]. Reduction is a means of reasoning that two executions α and α' are equivalent, where α' differs from α in having commuted two different threads' events. That two events may be commuted is justified by the fact that one is either a *left mover* or a *right mover*, defined as follows for our setting.

Definition 2 (Right mover)**.** *Operation o is a* right mover iff *for all $\alpha, a, b, \alpha', \omega, \omega', \rho, \rho'$, if α, a, b, α' is an execution with $\omega, \rho \vdash_0 \alpha, a, b, \alpha' \rightsquigarrow \omega', \rho'$, $op(a) = o$ and $tid(a) \neq tid(b)$, then α, b, a, α' is also a valid execution with $\omega, \rho \vdash_0 \alpha, b, a, \alpha' \rightsquigarrow \omega', \rho'$. I.e., we can move op. a right in the trace, swapping it with b.*

Definition 3 (Left mover)**.** *A* left mover *has the same definition as a right mover (Def. 2), replacing the precondition $op(a) = o$ with $op(b) = o$. I.e., we can move b left in the trace, swapping it with a.*

Given these definitions, we can prove that various operations are left movers, right movers, or both.

Lemma 4 (Movers)**.**

1. ownEx *and* ownRd *operations are right movers.*
2. relEx *and* relRd *operations are left movers.*
3. rd *and* wr *operations are both left and right movers.*

The proof of this Lemma is given in the Appendix.

Whether synchronization operations $\mathsf{sync}(t, op, x, v_1 ... v_n)$ are movers depends on their semantics. Under a standard multi-threading semantics, semaphore operations *wait* and *signal* are right- and left-movers, respectively [16]. Because synchronization operations on variables x require $\omega(x) = \top$, and $\top$-ownership of x must be invariant for an entire trace, commuting synchronization operations with other operations has no effect on ownership maps, so *wait* and *signal* are right- and left-movers in our setting as well. Many other synchronization operations are also movers; e.g., locking a reader/writer lock, or a mutex, is right mover, while releasing a lock or mutex is a left mover.

With these definitions in mind, we can define the conditions under which a thread's actions can be serialized. First, we define the notion of a sub-trace:

$$\begin{array}{lll} sub(\cdot, t) & = \cdot & \\ sub((a, \alpha'), t) & = a, sub(\alpha', t) & \text{where } t = tid(a) \\ sub((a, \alpha'), t) & = sub(\alpha', t) & \text{where } t \neq tid(a) \end{array}$$

With this, we can prove that sub-traces adhering to a certain form are serializable:[1]

Lemma 5 (Serializable sequences)**.** *Given an execution α such that $\omega, \rho \vdash_0 \alpha \rightsquigarrow \omega', \rho'$, if the sub-trace $sub(\alpha, t)$ defines a sequence of right movers followed by a sequence of left movers, then there exists an execution $\alpha', sub(\alpha, t), \alpha''$ that is equivalent to α and is valid such that $\omega, \rho \vdash_0 \alpha', sub(\alpha, t), \alpha'' \rightsquigarrow \omega', \rho'$.*

Proof. (Sketch) The equivalent execution can be constructed by repeatedly moving t's right-movers to the right in the trace, swapping them with operations of other threads, and likewise moving t's left-movers to the left in the trace, until they both meet in the middle, leaving all of t's events serial. □

[1] We could allow at most one "non-mover" in between the sequences of right-movers and left-movers, but non-movers never arise in valid traces.

Definition 6 (Two-phase ownership)**.** *A trace α exhibits* two-phase ownership *if for all a, a' such that $\alpha \equiv \alpha_0, a, \alpha_1, a', \alpha_2$ (where α_i could be empty), $a = \mathsf{relEx}(t, x)$ or $\mathsf{relRd}(t, x)$ for some x implies $a' \neq \mathsf{ownEx}(t, y)$ or $\mathsf{ownRd}(t, y)$ for all y. In other words, no ownership operations may follow release operations by the same thread. Any synchronization operations in the trace should be right movers up until no later than the first release operation, then followed by left movers.*

Theorem 7 (Validity implies serializability)**.** *A valid trace α in which thread t's events exhibit two-phase ownership implies that t's events are serializable.*

Proof. This follows easily from Lemma 5, since $sub(\alpha, t)$, in being valid and adhering to two-phase ownership, is sure to be a series of right movers followed by left movers. □

Our system does not enjoy a strong completeness property. For example, given an execution trace α devoid of any ownership operations, we might wish to prove that we can rewrite the trace to an equivalent one exhibiting two-phase ownership if α is serializable (where we consider all of a given thread's events as part of single transaction). But it is easy to see that this is not the case. Consider the following simple trace (where we have used pseudocode rather than lower-level events, for clarity):

Thread 1	Thread 2
`lock(m);`	
`tmp = x;`	
	`y = x;`
`x = tmp+1;`	
`unlock (m);`	

This trace is serializable (e.g., Thread 2's operation can be moved to the start of the trace), as indeed are all possible interleavings of these two transactions. But annotating this code with two-phase ownership would force `x` to be owned by Thread 1, and therefore signal an ownership violation when Thread 2 accesses it. Thus there is a race on `x`; we conjecture that two-phase ownership is sufficient to specify serializable transactions that do not contain races.

3.3 Relaxed Validation

If we imagine implementing basic validation with an on-line monitor, then the rules in Figure 6 imply that ownership checks (the premises of the rules) must be performed atomically with an action, e.g., a read or write. Likewise, ownership acquirement and release operations are assumed to check and update the ownership maps atomically. A straightforward way to ensure atomicity is to use locking. For example, prior to performing a read of variable x, thread t acquires a lock, checks the read ownership map, and then performs the read of x.

Implementing ownership checking this way could add significant overhead. While we must still be careful that accesses and modifications to ownership maps are thread-safe, this subsection shows that we do not need ownership checks to occur indivisibly when performed as part of read, write, or release operations. In particular, we prove that *relaxed trace validation*—in which ownership checks are proper operations separate from reads, writes, etc.—is sound and complete with respect to basic trace validation. Thus we are able to reduce the overhead of performing ownership checking without reducing its utility.

Figure 7 depicts the relaxed validation judgment. All of the operations that we had before are unchanged, but we add separate operations $\mathsf{ownWr?}(t, x)$ and $\mathsf{ownRd?}(t, x)$ for checking that t owns x exclusively (for writing), or for reading, respectively. The relaxed checking semantics permits read, write, sync, and release operations to be validated unconditionally (according to rules (CX-Read), (CX-Write), (CX-Sync), (CX-RelEx), and (CX-RelRd)),

[CX-READ]
$\omega, \rho \vdash \mathsf{rd}(t, x, v) \rightsquigarrow \omega, \rho$

[CX-WRITE]
$\omega, \rho \vdash \mathsf{wr}(t, x, v) \rightsquigarrow \omega, \rho$

[CX-RELEX]
$\omega, \rho \vdash \mathsf{relEx}(t, x) \rightsquigarrow \omega[x \mapsto \bot], \rho[x \mapsto (\rho(x) \setminus \{t\})]$

[CX-RELRD]
$\omega, \rho \vdash \mathsf{relRd}(t, x) \rightsquigarrow \omega, \rho[x \mapsto (\rho(x) \setminus \{t\})]$

[CX-SYNC]
$\omega, \rho \vdash_0 \mathsf{sync}(t, op, x, v_1 ... v_n) \rightsquigarrow \omega, \rho$

[CX-OWNEX]
$$\frac{\omega(x) = \bot \qquad \rho(x) = \emptyset}{\omega, \rho \vdash \mathsf{ownEx}(t, x) \rightsquigarrow \omega[x \mapsto t], \rho[x \mapsto \{t\}]}$$

[CX-OWNRD]
$$\frac{\omega(x) = \bot}{\omega, \rho \vdash \mathsf{ownRd}(t, x) \rightsquigarrow \omega, \rho[x \mapsto \rho(x) \cup \{t\}]}$$

[CX-CHECKWR]
$$\frac{\omega(x) = o}{\omega, \rho \vdash \mathsf{ownWr?}(o, x) \rightsquigarrow \omega, \rho}$$

[CX-CHECKRD]
$$\frac{t \in \rho(x)}{\omega, \rho \vdash \mathsf{ownRd?}(t, x) \rightsquigarrow \omega, \rho}$$

[CX-TRACE]
$$\frac{\omega, \rho \vdash a \rightsquigarrow \omega', \rho' \qquad \omega', \rho' \vdash \alpha \rightsquigarrow \omega'', \rho''}{\omega \vdash a, \alpha \rightsquigarrow \omega'', \rho''}$$

where

$o \in \mathit{Owners} ::= t \mid \top$
$b \in \mathit{OwnOper} ::= \ldots \mid \mathsf{ownWr?}(o, x) \mid \mathsf{ownRd?}(t, x)$

Figure 7. Relaxed trace validation

but ownEx and ownRd events are checked as before, and the separate ownership checks are checked similarly, according to (CX-CheckWr) and (CX-CheckRd).

In this setting, we are presuming the program has been changed to emit a separate ownership test prior to each read, write, sync, and release operation, while ownership operations remain atomic—checking the conditions for allowing ownership and acquirement of ownership happen indivisibly. We formalize this assumption below.

Equivalence to basic validation. We can prove that the relaxed validation is sound and complete with respect to the basic validation, and as such that validity in the relaxed semantics implies serializability. To establish this requires a means to relate traces in the basic and relaxed settings. First we define a way to strip explicit ownership checks for a relaxed-setting trace, resulting in a trace that can be subject to basic validation:

$$\begin{array}{ll} [\cdot] = \cdot & \\ [a, \alpha'] = [\alpha'] & \text{where } op(a) \in \{\mathsf{ownWr?}, \mathsf{ownRd?}\} \\ [a, \alpha'] = a, [\alpha'] & \text{otherwise} \end{array}$$

Conversely, we define a way to add explicit ownership checks to a trace that lacks them:

$$\begin{array}{ll} \langle\cdot\rangle = \cdot & \\ \langle a, \alpha'\rangle = \mathsf{ownWr?}(tid(a), var(a)), a, \langle\alpha'\rangle & \\ \qquad \text{where } op(a) \in \{\mathsf{wr}, \mathsf{relEx}\} & \\ \langle a, \alpha'\rangle = \mathsf{ownWr?}(\top, var(a)), a, \langle\alpha'\rangle & \text{where } op(a) = \mathsf{sync} \\ \langle a, \alpha'\rangle = \mathsf{ownRd?}(tid(a), var(a)), a, \langle\alpha'\rangle & \\ \qquad \text{where } op(a) \in \{\mathsf{rd}, \mathsf{relRd}\} & \\ \langle a, \alpha'\rangle = a, \langle\alpha'\rangle & \text{otherwise} \end{array}$$

Finally, we define the well-formedness judgment $C \vdash_{wf} \alpha$ to formalize the requirement that all read, write, release, and sync actions in α are preceded by an ownership check. The rules for this judgment appear in Figure 8. Here, the *check map* C maps triples (t, x, p) to either ✓ or ×, with $p \in \{r, w\}$. We write $C_\times$ to denote the map which maps all triples to ×. In the rules, $C(t, x, p) = ✓$ implies that thread t has previously checked that it owns x, either for reading if p is r or exclusively if p is w. Thus rule [WF-CheckRd] sets $C(t, x, r)$ to ✓ when checking the remainder of the trace upon seeing action $\mathsf{ownRd?}(t, x)$, while rules [WF-Rd] and [WF-RelRd] each require that t has previously performed a read check. Though trace well-formedness is superficially similar to trace validation, we emphasize that it only guarantees that a thread t precedes accesses to x with an appropriate check; this check is by no means guaranteed to succeed.

The rules for writes and sync operations are similar to those for reads (cf. rules [WF-CheckWr], [WF-Sync], [WF-Wr] and [WF-Rel]). Rules [WF-Own] and [WF-OwnRd] treat their respective ownership operations similarly to ownership checks since ownership checking is part of the semantics of ownEx and ownRd (cf. rules [CX-OwnEx] and [CX-OwnRd] in Figure 7). Likewise, operations $\mathsf{relEx}(t, x)$ and $\mathsf{relRd}(t, x)$ invalidate any prior checks made by t for variable x.

Optimized checking. While it is straightforward to see that for all basic traces α that $C_\times \vdash_{wf} \langle\alpha\rangle$—i.e., that it admits traces in which all operations by thread t are preceded by a check in t's subtrace—the well-formedness judgment admits even more optimized traces. For example, in the trace $\mathsf{ownEx}(t, x), \alpha', \mathsf{ownWr?}(t, x), \alpha'', \mathsf{wr}(t, x, v)$, where no event a in α' or α'' has $tid(a) = t$, we can safely remove the $\mathsf{ownWr?}(t, x)$ event from the trace. This is because ownership is invariant—once a thread t acquires ownership of some variable x, no other thread can change x's ownership to not include (or be) t. This implies that if the $\mathsf{ownEx}(t, x)$ event in the above trace is valid, then the $\mathsf{wr}(t, x, v)$ also will be (assuming the intervening traces α, α' are valid as well). We have both $C_\times \vdash_{wf} \mathsf{ownEx}(t, x), \alpha', \mathsf{ownWr?}(t, x), \alpha'', \mathsf{wr}(t, x, v)$ and $C_\times \vdash_{wf} \mathsf{ownEx}(t, x), \alpha', \alpha'', \mathsf{wr}(t, x, v)$.

The bottom of Figure 8 also defines a compatibility judgment between a check map C and ownership maps ω and ρ, for purposes of establishing proper inductive hypotheses in the proofs.

Now we can prove soundness and completeness.

Lemma 8 (Soundness)**.** *If* $\omega; \rho \vdash \alpha$ *where* $C \vdash_{wf} \alpha$ *and* $\omega; \rho \vdash C$ *then* $\omega; \rho \vdash_0 [\alpha]$.

Lemma 9 (Completeness)**.** *If* $\omega; \rho \vdash_0 \alpha \rightsquigarrow \omega'; \rho'$ *then* $\omega; \rho \vdash \langle\alpha\rangle \rightsquigarrow \omega'; \rho'$.

The proofs are by straightforward induction on the given derivations. Note that we cannot prove the more symmetric completeness result, i.e., $\omega; \rho \vdash_0 [\alpha] \rightsquigarrow \omega'; \rho'$ implies $\omega; \rho \vdash \alpha \rightsquigarrow \omega'; \rho'$. As a counterexample, for trace $\alpha \equiv \mathsf{ownRd?}(t, x), \mathsf{ownEx}(t, x), \mathsf{rd}(t, x, v)$ we cannot prove $\omega_\bot; \rho_\emptyset \vdash \alpha$ but we can prove $\omega_\bot; \rho_\emptyset \vdash_0 [\alpha]$.

Theorem 10 (Relaxed Validity Implies Serializability)**.** *For a relaxed-semantics trace* α *in which thread* t*'s events exhibit two-phase ownership,* $\omega; \rho \vdash \alpha$ *and* $C_\times \vdash_{wf} \alpha$ *implies* t*'s events are serializable.*

4. Implementation

This section describes the implementation of our dynamic ownership analysis. There is a runtime library with definitions for the ownership assertions and wrappers for library functions, and a compiler that adds memory access checks to the code. The compiler uses a simple static analysis to elide access checks that can be

$$\text{[WF-RD]}\quad \frac{C(t,x,r) = \checkmark \quad C \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{rd}(t,x,v),\alpha} \qquad \text{[WF-WR]}\quad \frac{C(t,x,w) = \checkmark \quad C \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{wr}(t,x,v),\alpha}$$

$$\text{[WF-REL]}\quad \frac{C(t,x,w) = \checkmark \quad C[(t,x,w) \mapsto \times][(t,x,r) \mapsto \times] \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{relEx}(t,x),\alpha}$$

$$\text{[WF-RELRD]}\quad \frac{C(t,x,r) = \checkmark \quad C[(t,x,r) \mapsto \times] \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{relRd}(t,x),\alpha}$$

$$\text{[WF-CHECKWR]}\quad \frac{C[(o,x,w) \mapsto \checkmark] \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{ownWr?}(o,x),\alpha} \qquad \text{[WF-CHECKRD]}\quad \frac{C[(t,x,r) \mapsto \checkmark] \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{ownRd?}(t,x),\alpha}$$

$$\text{[WF-SYNC]}\quad \frac{C(\top,x,w) = \checkmark \quad C \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{sync}(t,op,v,x),\alpha}$$

$$\text{[WF-OWN]}\quad \frac{C[(t,x,w) \mapsto \checkmark][(t,x,r) \mapsto \checkmark] \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{ownEx}(t,x),\alpha}$$

$$\text{[WF-OWNRD]}\quad \frac{C[(t,x,r) \mapsto \checkmark] \vdash_{wf} \alpha}{C \vdash_{wf} \mathsf{ownRd}(t,x),\alpha} \qquad \text{[WF-EMPTY]}\quad C \vdash_{wf} \cdot$$

$$\text{[WF-COMPATIBILITY]}\quad \frac{\begin{array}{c}\forall t,x,o.\ C(t,x,r) = \checkmark \Rightarrow t \in \rho(x) \wedge \\ C(o,x,w) = \checkmark \Rightarrow o = \omega(x)\end{array}}{\omega;\rho \vdash C}$$

where

$$\begin{array}{rcl} p \in RWop & ::= & r \mid w \\ C \in WFmap & ::= & Owners \times Var \times RWop \to \{\times, \checkmark\} \end{array}$$

Figure 8. Well-formed, optimized execution traces

proven not to fail. We used the Phoenix framework [18] to implement a compiler for C and C++ programs running on 32-bit x86 processors. We start by explaining the access checks and runtime library and then we describe the static analysis.

4.1 Memory ownership states and data structures

Figure 9 shows the full transition diagram for memory ownership states in our implementation, expanding the simpler diagram shown in Figure 1. We record the ownership state of memory using two data structures: the *ownership table* and the *cluster table*.

The ownership table maintains one byte of state for each eight byte slot in virtual memory. This byte encodes *not accessible*, *not owned*, *read-only*, *unchecked*, a thread identifier when the slot is owned exclusively by a thread, or a cluster identifier if the slot has been given to a cluster. Thread identifiers are allocated dynamically when threads start and are freed when they exit. Cluster identifiers are allocated and freed dynamically from the same space.

The virtual memory for the ownership table is reserved by our runtime library at program startup time and we install a page fault handler to allocate physical pages to the table on demand. Therefore, this table introduces a space overhead of approximately 12.5%

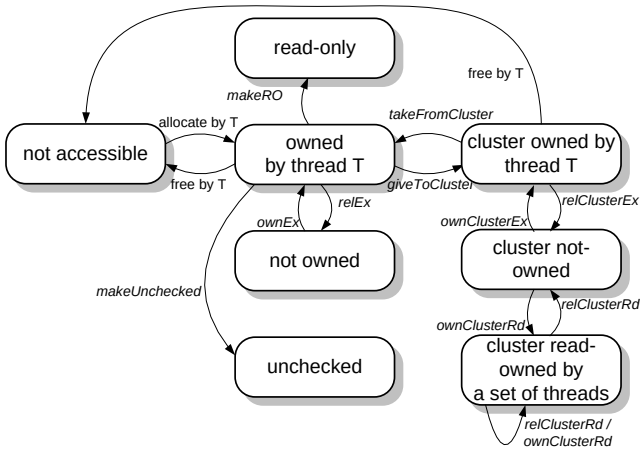

Figure 9. Memory ownership states.

while supporting up to 250 threads and clusters. The compiler aligns global and local variables on eight-byte boundaries to ensure that different variables can have different ownership states (as in [1]). The standard memory allocators also align heap allocations on eight byte boundaries. But if different fields in the same structure can have different ownership states, the programmer must use alignment pragmas to ensure they are in different slots.

The cluster table records the ownership state for clusters. It has a fixed size of 256 entries so that it can be indexed efficiently using a cluster identifier. Each entry has a 4-byte status word and a bitmap with 256 bits. The status word is null if the cluster is not owned; it records a thread identifier if the cluster is owned exclusively by a thread or records a count of the number of readers (`ored` with 0x80000000) if the cluster is owned for reading by a set of threads. The bitmap tracks each thread that owns the cluster for reading. The cluster table introduces a fixed space overhead of 9KB.

As shown in Figure 9, we support setting individual memory slots to the read-only state (which allows any thread to read) but we do not support the read-owned state unless a slot is given to a cluster. We made this decision to keep the space overhead low because supporting this state with up to 250 threads for individual memory slots would increase the space overhead from 12.5% to 400%. As mentioned in Section 2, the *unchecked state* is used to label synchronization variables or data with otherwise benign races: any thread can read or write unchecked memory.

The read-only and unchecked states are sticky: the state of a memory slot cannot change once it is set to read-only or unchecked, which implies the memory cannot be freed (which would require moving it to the not-accessible state). Otherwise, accesses through dangling references could allow threads to read or write data owned by other threads, which would violate our modular soundness property. Therefore we do not allow stack variables to be in these states and we delay frees of memory in one of these states until the program terminates.

These restrictions may seem overly burdensome but they are easy to overcome using custom allocators for objects with read-only or unchecked fields. For example, our implementation will delay freeing a queue from Figure 2 to the global heap because the queue has read-only fields that point to synchronization objects. But nothing prevents the programmer from writing a custom allocator that implements `free` by adding freed queues to a list and `malloc` by reusing a queue from the list (or allocating a new one if the list is empty). Reusing a queue does not violate the ownership policy because it requires re-initializing the fields that are not read-only (`size`,`occupied`,`nextin`,`buf`) but it does not require changes

to the read-only fields that point to synchronization variables. Since we believe this is a common pattern, we provide a generic implementation in our runtime library. The programmer can create one of these allocators for an object type by specifying a function to allocate objects when the list is empty (like queueInit in our example) and a function to reuse objects from the list.

4.2 Compiler

The compiler inserts ownership checks before read and write accesses to memory. These checks consult the memory ownership and cluster tables to determine whether to allow the access. As we prove in Section 3.3, a check and the memory access it precedes do not need to happen indivisibly. This allows us to use efficient code sequences to implement the checks, which is important because they are executed frequently. Figure 10 shows an example code sequence that implements a read check. In this example, the address of the memory about to be read is in the eax register. The sequence starts by loading the state of the slot pointed to by eax from the memory ownership table into al. Then it checks if the running thread owns the slot exclusively by comparing the state with the thread's identifier. The thread identifier is stored in per-thread memory at thread creation time and is loaded into a stack variable at the start of each function. The second check determines if the slot is in the read-only state (encoded as 8). If either of these checks succeeds, the read is allowed. Otherwise, the sequence calls the __slowPathR_AL function to check if the slot was given to a cluster the thread owns or is in the unchecked state. This "slow path" check is also efficient: the function expects the ownership state in register al and it uses pre-computed information that is stored in per-thread memory to access the bitmap in the cluster table without using other registers. If the slow path fails, we execute the x86 instruction **int** 3, which signals a failure by triggering a breakpoint; otherwise, we allow the read. The other code sequences are similar. All are short, use a single register, and are inserted in an early compilation phase within Phoenix to expose them to further optimizations.

```
1       shr eax,3
2       movzx eax,byte ptr [eax+40000000h]
3       cmp al,byte ptr [esp+4Ch]
4       je L1
5       cmp al,8
6       je L1
7       call __slowPathR_AL
8   L1:
```

Figure 10. Code sequence that implements ownRd?.

In addition to ownership checks, the compiler inserts code to update the memory ownership table on function entry to grant exclusive ownership of arguments and local variables to the running thread. The function epilogues are modified to revert this memory to the not-accessible state after checking that it is owned exclusively by the running thread. These checks prevent deallocation of memory that may be in use by other threads. We remove most of these checks with a simple static analysis (Section 4.4).

We implement *control-flow integrity* as in [5] to prevent memory errors from bypassing our checks. We assign a special ownership value to the start of functions whose address is taken, and use a check similar to the one in Figure 10 to ensure that indirect calls only target these functions. We prevent errors from subverting other indirect control flow transfers by ensuring that the memory locations with their targets are always in the not-accessible state; for example, return addresses are always in the not accessible state.

The compiler also inserts *guard objects* adjacent to global variables and those local variables whose address is taken (as in [1]). These guard objects are always in the not-accessible state. While guard objects are not strictly necessary, with them we can efficiently detect sequential buffer overflows and underflows, which are a common exploitable error.

4.3 Runtime library

The runtime library implements the ownership assertions and wrappers for library functions. The ownership assertions are listed next to the edges in Figure 9 and they can only be called in the states shown in the figure. The implementation of these functions is straightforward: they first check that the memory or cluster they receive as an argument is in the appropriate state and then they modify the state in the memory or ownership table according to the figure. As we prove in Section 3.3, the check and modification do not need to be atomic except for ownEx, ownClusterEx, and ownClusterRd.

We implement ownEx by performing a sequence of atomic cmpxchg instructions on the affected ownership table entries. As an optimization, our implementation performs cmpxchg on four entries at a time when possible. If any of the cmpxchg instructions fails to update the table because the state is no longer unowned, the function triggers a breakpoint. ownClusterEx uses cmpxchg on the cluster status to set it to the thread identifier. ownClusterRd uses cmpxchg to increment the count of readers in the cluster status and an atomic bit manipulation instruction (btc) to update the cluster's bitmap (relClusterRd is implemented in a similar way because several threads can acquire and release the cluster for reading concurrently).

We define wrappers for common library functions and system calls. These wrappers perform ownership checks on the memory that is accessed by the functions they wrap. For example, the wrapper for memcpy checks that the source buffer is readable and that the destination buffer is writable. The compiler replaces direct and indirect calls to the original functions with calls to the wrappers.

The wrappers for malloc and free are the most interesting. The wrapper for malloc sets the ownership table entries for the allocated memory to the identifier of the calling thread. In addition, it sets the entry corresponding to the allocator metadata to a special *heap guard* state. This state is used to prevent freeing of pointers to non-heap memory or to the middle of heap allocations. Threads executing instrumented code cannot read or write memory in the *heap guard* state. Therefore, this also prevents sequential buffer overflows and underflows that cross heap allocation boundaries.

The wrapper for free checks that the slot before the memory being freed is in the *heap guard* state and that the memory being freed is owned exclusively by the calling thread. Frees of memory blocks that contain slots in the read-only or unchecked states are delayed until the end of the execution. Attempts to free memory in other states raise an error. This implementation ensures that incorrect heap management cannot violate the ownership policies.

To prevent delaying frees from causing unbounded memory leaks, we enforce the following restriction: after free is called on memory in the read-only or unchecked state, calls to makeRO or makeUnchecked signal an error.

4.4 Static Analysis

As discussed in Section 3.3, it is not necessary to insert ownership checks before every memory access because ownership is invariant: once memory becomes owned by a thread t, no other thread can revoke that ownership. Similarly, memory in the read-only or unchecked states can never be made otherwise. As such, if an access to location x is preceded by an acquirement or check of the needed ownership of x, and there is no intervening release, we can elide the check. We use static analysis to elide these checks.

The basic idea is to use an intraprocedural dataflow analysis to compute the set of memory locations that must be owned exclu-

sively, or for reading only, by the current thread at a given program point. Reads or writes to locations that are covered by these *known-exclusive* sets, or reads covered by the *known-readable* sets, do not require any checks. Ignoring the effects of function calls for the moment, locations are added to the known-exclusive set (gen) for each ownEx, ownClusterEx, and write, and they are removed (kill) for each free, relEx, relClusterEx, and giveToCluster. Gen and kill functions for the known-readable set are similar, and in both cases sets are intersected at joins in the control-flow graph. We also allow the programmer to write assertOwnedEx and assertOwnedRd to force checks on particular memory locations, which add locations to the known-exclusive and known-readable sets, respectively. These assertions can help reduce the total number of checks by effectively hoisting checks above complicated control flow that might otherwise foil the check-elimination analysis.

There are two further complications in the analysis. First, building the sets above requires memory locations to be unambiguously identified when adding them to the set, which is complicated by aliasing and pointer arithmetic. Second, we must account for the possible effects on the ownership state of intervening function calls. We handle these issues fairly simply, to avoid an expensive whole-program analysis.

For the first case we also run a variant of the analysis that keeps track not of sets of memory locations, but of sets of symbolic (pointer,length) pairs. The intuition is that if we insert a check for a write to $*$p, we may be able to elide the check for a second write through the same pointer. We must however be careful that what p points to did not change in between the two calls. To ensure this, the kill function will remove a pair if the location of either symbol leaves the known-exclusive set or when a write may overlap with the location holding the symbol itself. We can elide checks from reads or writes through a pointer+offset as long as the pointer is in the set and $0 \leq \text{offset} < \text{length}$. Both offset and length may be symbolic expressions: we use a simple symbolic evaluation to determine, for example, that symbolic offset n is less than symbolic length $n + 1$, if we know that $n + 1$ is not an overflow. There is a similar analysis for eliding read checks.

To avoid the pessimistic assumption that a function could release ownership of any memory location, we perform a simple bottom-up interprocedural analysis to summarize the "release-behavior" of each function, tracking the set of locations that may be released from exclusive (or freed) and the set of locations that may be released from read. Thus, when the intraprocedural analysis encounters a call to a function, it uses the summary to adjust the known-readable and known-exclusive sets appropriately. We do not summarize the "acquire-behavior" of functions, though we could do so to improve precision.

5. Evaluation

We used seven C and C++ programs to evaluate our dynamic ownership checking tool. We started by annotating the programs with ownership policies and assessing the required changes. Then we ran experiments to measure the time and space overheads introduced by our tool. This section presents our results and discusses several concurrency and memory errors that we found.

5.1 Programs, annotations, and bugs

We annotated seven multi-threaded programs: pfscan, aget, pbzip2, stunnel, genome, ctrace, and nullhttpd. We chose the first four programs to facilitate a comparison with SharC [2]. These four programs synchronize threads using mutexes, condition variables, and thread joins. We also chose programs that use different synchronization primitives to demonstrate the generality of our ownership assertions: genome, taken from the STAMP benchmark suite [4], uses barriers and ctrace uses semaphores. The nullhttpd Web server is known to be buggy. We chose nullhttpd to evaluate our tool's ability to find bugs.

	lines	changes		annotations	
aget	1098	6	0.55%	11	1.00%
ctrace	1408	8	0.57%	33	2.34%
genome	9645	56	0.58%	63	0.65%
nullhttpd	3030	3	0.00%	10	0.33%
pbzip2	15188	21	0.14%	81	0.53%
pfscan	1073	22	2.05%	40	3.73%
stunnel	235366	94	0.04%	14	0.01%
total	266808	210	0.08%	252	0.09%

Table 1. Change and annotation counts.

We annotated the programs using the following methodology. First, we ran the programs with no annotations using our tool. Since the default ownership policy prevents sharing, the tool signaled an error whenever a variable was shared in one of these executions. Then we inspected the source code to understand how these variables should be shared and we inserted annotations to enforce the appropriate ownership policy. We repeated this process with different test inputs until the tool stopped signaling errors. In several cases, we inserted annotations right next to matching comments in the code but our analysis showed that some of these comments were wrong. Concurrent programs are subtle and specifying and enforcing the right ownership policy without the support of a tool like ours can be tricky. We found that our incremental methodology works well: it allows programmers to debug the program and ownership policies together.

Table 1 tabulates the total lines of code we changed or annotated for each program. Since we maintain ownership state for eight-byte memory slots, a common change was the addition of alignment pragmas to structure definitions to ensure that different fields are in different slots. These pragmas are only needed if different fields in the same structure can have different ownership states.

The annotation count is small compared with the size of the annotated programs. In the worst case, we changed 6% of the lines in a benchmark but we changed less than 0.2% of the lines in total. SharC [2] has lower annotation counts in the first four benchmarks but it provides weaker guarantees; in particular, it will fail to catch problems due to memory safety violations. We obtained versions of aget, pbzip2, and pfscan with both SharC and Deputy [7] annotations from the authors of SharC. We found that adding Deputy [7] annotations to achieve spatial memory safety increases their annotation counts to be greater than ours on average.

We found eight bugs in the benchmark programs using our tool: three serious races and five buffer underflows. Next, we describe the benchmarks, the annotations, and the bugs in more detail.

The pfscan program searches for strings in files. The main thread finds the paths of all files that need to be searched and inserts them in a queue. Worker threads loop taking a path from the queue and searching the corresponding file. This queue is very similar to the one in Figures 2 and 3, and requires similar annotations. In addition, the main thread uses a shared variable protected by a mutex to synchronize with the workers on completion. We inserted ownEx annotations for this variable after the mutex is acquired and relEx before it is released. The rest of the annotations are for variables that are initialized by the main thread and are read-only for the rest of the execution. We inserted makeRO annotations for these variables right after their initialization.

Aget parallelizes file downloads. The main thread obtains the size of the file and initializes an array with structures describing disjoint byte ranges. Then it forks a worker thread to download each range and waits to join them. We inserted a relEx annotation for the structure describing the range before each fork and a corresponding

ownEx annotation after each join. We also added a ownEx annotation for the structure at the start of the worker routine and a corresponding relEx annotation at the end. These annotations acquire and release ownership of all the fields in the structure except for the field with the thread identifier. This field is owned exclusively by the main thread throughout the execution to allow its use as an argument to the join. Our tool found a benign race on a variable used to print a progress bar. We added an annotation makeUnchecked to make this variable unchecked. Our tool found a buffer underflow in aget that can cause the program to read a byte before the start of a string.

Pbzip2 is a parallel implementation of the bzip2 compression algorithm. A producer thread reads blocks from a file and places a structure describing each block in a queue. Several consumer threads take blocks from the queue and compress (or decompress) them. The consumers put compressed (or decompressed) blocks in an array that is protected by a mutex. A writer thread scans this array and writes the blocks to disk. The annotations that we inserted for the queue and the shared array are similar to those in pfscan. Additionally, we inserted annotations to mark variables read-only and to mark a variable with a benign race unchecked.

The stunnel program provides a tunneling service for TCP over SSL. The main thread accepts connections in a loop and forks a client thread to manage each connection. Most of the data is shared read-only between threads or is thread private. We inserted makeRO annotations to mark this data read-only after initialization and our default annotations were sufficient for thread private data. Stunnel uses the OpenSSL library to perform encryption. We also instrumented this library.

The genome benchmark program takes a large number of DNA segments and matches them to reconstruct the original source genome [4]. We modified the original benchmark, written to use software transactional memory, to use locks instead. The algorithm runs in several phases with barriers between them. The first phase uses a hash set to create a set of unique segments from the initial segment pool. We associate a cluster with the hash set. We insert a single ownClusterEx before updating the hash set and a matching relClusterEx after. After the barrier at the end of the first phase, we insert an ownClusterRd for the cluster because the hash set is read-only for the rest of the execution. This is another example of dynamic change in ownership policy. In the second phase of the algorithm, threads match unique segments using shared arrays and hash tables. The third phase computes the final sequence. The annotations that we inserted in the code for these phases resemble ones already described.

The ctrace library provides tracing functionality for multithreaded programs. The key data structure is a hash table with per-thread information. The hash table is protected with a read-write lock implemented using semaphores. We associated a cluster with the hash table and we inserted annotations in the read-write lock implementation to take and release exclusive or read ownership of the cluster as appropriate. We used the makeUnchecked annotation to deal with several benign races.

The nullhttpd Web server has a concurrency pattern similar to the one in stunnel: the main thread accepts connections in a loop and forks a client thread to handle each connection. There is an array of per-connection data structures. Client threads receive a pointer to one of these entries initialized by the main thread and they mark this entry as not in use when they exit. The main thread scans this array looking for free entries. Client threads exit when their connection is closed or is idle for more than a timeout. The annotations we inserted in this program are similar to those used in aget except that we marked one of the fields in each connection data structure unchecked because it has a benign race.

Our tool found three serious concurrency bugs in nullhttpd. The first is a race on a handle field in the per-connection data structure. The main thread initializes a thread handle field in this structure after it creates a thread. A client thread may close an invalid handle if it accesses this field before it is initialized by the main thread. The second bug is a race on the per-connection data structure that can lead to a double free. Before exiting, a client thread frees data pointed to by its connection data structure and uses memset both to zero the pointers to the freed data and to signal main that the data structure is no longer in use. This can cause the main thread to free the data again if it reuses the data structure before the pointers are zeroed by the client thread. The last bug is a race on a static variable used to implement a readdir-like function in the Windows version of nullhttpd. The tool also found four buffer underflows in nullhttpd that can cause the program to write data to bytes before the beginning of a buffer.

We found the bugs described in this section by running the benchmarks in the next section and simple tests to increase code coverage. It was not necessary to explore different thread schedules to uncover these bugs. Dynamic ownership analysis can significantly increase the set of schedules that uncover a bug. For example, our analysis uncovers the last bug we described in any execution where a thread other than main accesses the static variable. Without our analysis, the bug has no adverse effects unless two threads use the readdir-like function concurrently. We would likely find more bugs by combining dynamic ownership analysis with tools like Chess [17] that explore different thread schedules systematically.

5.2 Performance measurements

We also measured the overhead introduced by our tool. We compiled the benchmarks with and without dynamic ownership analysis. We used the Phoenix [18] compiler with the -O2 (maximize speed) option in both cases. Phoenix is a production-quality compiler—for example, it achieves SPEC Int 2000 performance within 10% of the C/C++ compiler shipped with Microsoft Visual Studio. Then we ran experiments comparing the elapsed time, throughput, and memory usage of the two versions. Memory usage is the peak working set size as reported by the Windows PSAPI interface. We averaged the elapsed time, throughput, and memory usage across at least 10 runs of each experiment. The standard deviation was below 3% of the computed average in all experiments. We report the percent increase in average elapsed time and memory usage and the percent decrease in average throughput.

All the experiments ran on HP xw4600 workstations with an Intel Core2 Duo CPU at 2.66 GHz and 4GB of RAM, running the Windows Vista Enterprise SP1 operating system. The workstations were connected with a Buffalo LSW100 100 Mbps switching hub for the experiments involving the network.

The experiments used to evaluate the first four programs attempt to reproduce the experiments described in [2]. To evaluate pfscan, we searched for the string "HELLO" in three copies of the PDF files from the proceedings of DSN 2005, 2007 and 2008. We eliminated I/O overhead by ensuring that the files fit in the operating system buffer cache and by warming up the cache before the experiment. We used aget to fetch a compressed file with 182MB from an idle Microsoft IIS server connected to the hub. The experiment to evaluate pbzip2 compressed a 4MB file. We evaluated stunnel by encrypting three connections to a simple echo server and measuring the time to send and receive 10,000 messages. We ran the genome benchmark with g=16E3, n=16E6, s=32 and t=2 and we ran the simple benchmark in the ctrace distribution with two threads that write 500000 messages to a trace file. We measured the throughput of nullhttpd with the default configuration (which limits the maximum number of simultaneous connections to 50) using

	time	space
aget	0.0%	0.0%
ctrace	27.0%	6.5%
genome	44.2%	12.5%
nullhttpd	0.0%	10.4%
pbzip2	49.0%	18.6%
pfscan	37.2%	14.0%
stunnel	20.8%	8.7%
average	25.8%	10.4%

Table 2. Overhead in time and space.

the apache benchmark. We increased the number of simultaneous clients until the throughput stopped increasing. The results reported were obtained with 30 simultaneous clients fetching a small Web page 10,000 times.

The middle column of Table 2 shows the percentage increase in elapsed time due to our dynamic ownership analysis (for nullhttpd it shows the percentage decrease in thoughput). The average overhead across all benchmarks is 26%. For memory intensive benchmarks like pbzip2, the overhead can be as high as 49%, which is probably too high for our tool to be used during production but low enough for use during testing.

The overhead is negligible for nullhttpd and aget. To investigate why, we measured the increase in CPU time due to our instrumentation in aget and nullhttpd. We used the GetSystemTimes function in Windows to measure the CPU time. We found that our instrumentation does not measurably increase the CPU time for aget because most of the CPU time is consumed executing operating system code that we wrap but do not instrument. Our instrumentation increases the CPU time for nullhttpd by 24%. We believe that these overheads are sufficiently low for our tool to be used during production to improve security and diagnosability of concurrency errors.

The rightmost column of Table 2 shows the memory overhead incurred by the various benchmarks. Since we have one byte in the ownership table for each eight byte memory slot, we would expect the space overhead introduced by our technique to be around 12.5%. The average space overhead across all the benchmarks is 10%. Values below the expected 12.5% are due to memory that is touched in libraries that we do not instrument and values above are due to updating ownership table entries for memory that is allocated but never touched in the benchmark (the operating system faults pages on demand for large allocations).

The time overhead of Shoal [3], the new version of SharC that improves expressiveness, seems to be similar to ours but Anderson et al. reported lower time overhead for pfscan, pbzip2, and stunnel using SharC [2]. Both Shoal and SharC provide weaker guarantees than our dynamic ownership analysis. In particular, they do not enforce sharing annotations in the presence of memory errors. Most of our overhead in these benchmarks is due to the checks to prevent ownership policy violations due to memory errors. Our static analysis can frequently prove that the target object for an access is owned without being able to prove that the access is within bounds. We believe that a more sophisticated static analysis would be able to close the performance gap. The space overhead of our technique is lower than SharC's on average despite the fact that we support up to 250 threads whereas SharC only supports up to seven to keep the space overhead low.

6. Related Work

Many static and dynamic analyses have been developed to detect concurrency-related problems. Static analyses, e.g., for ensuring race-freedom [19] or atomicity [12], have the advantage that they cover potentially all program executions, but the disadvantage that they often have many false alarms or fail to scale. Dynamic analyses such as ours have the opposite properties. The coverage problem can be mitigated by employing active testing tools, such as CalFuzzer [15], or employing hybrid static/dynamic analysis [6]. For brevity, we focus predominantly on dynamic analyses, organized according to the concurrency problem detected.

Data races are a well-known class of error in which two threads access the same memory location without using appropriate synchronization, and at least one access is a write. Many dynamic analyses for detecting races have been developed, from imprecise *lockset-based* detectors such as Eraser [21], which work by ensuring that a shared location is always accessed with a particular lock held, to precise detectors such as Goldilocks [8] and FastTrack [10], that track some form of *happens-before* relation. We could implement a dynamic data race detector with our system by wrapping every read and write with ownership assertions for reading and exclusive access, respectively. Though useful, the lack of data races is not sufficient to ensure correctness. Protecting each individual access by a lock, for example, trivially makes any program data race-free, but concurrency errors would remain. Moreover, data races are not always problematic; programs often exhibit so-called *benign races*. On the other hand, data race detection tools are easy to use, requiring no direct input from the programmer.

Atomicity violations are a better indicator of concurrency errors. Using a tool such as Velodrome [11], Wang et al. [22], or the Atomizer [9], a programmer may specify whether a method or code block should execute atomically, and the tool verifies that it does so. Velodrome is both sound and complete: atomicity violations occurring within a given execution are precisely flagged; Wang et al. is similarly very accurate. Atomizer is nearly sound (it admits some data races), and is incomplete in that some atomic executions are erroneously flagged; simplistically, this can be explained by the fact that some serializable executions may have data races. When used to enforce serializability, our system is sound (see Theorem 10), but is similarly incomplete. Atomizer is also limited in that it is lockset-based, whereas our approach is indifferent to the synchronization strategy used in the program. On the other hand, atomicity specifications are higher-level than our ownership assertions, and therefore may be easier to use.

Sharing violations. Our tool is one of several that consider enforcement of data-centric *sharing policies*, which in some cases may imply properties like atomicity.

Using SharC [2] and its successor Shoal [3], programmers annotate standard types with one of five *sharing qualifiers*. For example, the qualifier private designates thread-local data, readonly indicates shared but read-only data, and locked(m) indicates shared data accessible only when lock m is held. Programmers can switch the sharing strategy using a *sharing cast* to assign data a different qualifier. This cast succeeds if the source pointer is *unique*, a restriction enforced by reference counting.

Compared to SharC, our specifications are conceptually simpler: to read or write a location, a thread asserts ownership of that location for reading or writing, respectively—the synchronization strategy is immaterial. Our simpler specifications also come with fewer restrictions: SharC and Shoal require some contortions to satisfy the unique pointer constraint and thus change the sharing strategy. In contrast, our ownership policies may be changed freely, as may be the means employed by the program to enforce them. On the other hand, SharC's higher-level specifications are more terse, leading to fewer annotations. SharC uses whole-program analysis to provide some compile-time guarantees, e.g., that data declared private is never accessed by multiple threads. As such, it relies on static analysis for soundness: if the analysis is too conservative and rejects a correct program, it would have to be modified before it

can be run at all. In contrast, conservative analysis in our system can only lead to missed optimization opportunities. SharC does not include the memory-safety checks needed for soundness. In our experience, these checks add overhead and remove optimization opportunities; one of the SharC authors suggested to us that combining SharC with Deputy to provide similar assurances could bring overheads closer to 2x. This combination would also increase the number of annotations to be greater than ours in the benchmarks we used in common.

Isolator [20] dynamically enforces a programmer-specified *locking discipline*, which states that a particular lock must be held to access particular shared data. Threads that adhere to this discipline are protected from interference by ill-behaved threads; this is similar to how data owned by a thread in our approach is protected from interference by other threads. Our approach is more general in not being tied to lock-based synchronization, though it is possible that our specifications are more verbose as a result.

Finally, Hammer et al. [14] use a dynamic tool to enforce *atomicity-set serializability*, a more inclusive property than traditional serializability. The idea is that certain data belong to an *atomic set* and should be updated atomically by designated *units of work* (UoW). As one UoW may invoke other UoWs during processing, it may be spuriously considered non-serializable, even though the nested UoWs are effectively independent. We conjecture we could enforce atomic-set serializability by asserting ownership of the appropriate atomic sets at the outset of a UoW, releasing ownership at its conclusion. Hammer at al. have found that atomic sets and units of work can be inferred effectively from the structure of class declarations, and we suspect that ownership annotations could be similarly inferred from class structure; writing ownership assertions by hand would be more verbose.

All these tools either provide weaker guarantees than ours, have higher overhead, or both. FastTrack's slowdown ranged from 0.9x to 14.8x for the benchmarks they considered; Goldilocks' was up to 11.4x given integrated VM support. Velodrome's slowdown is between 1.1x and 70x, and Atomizer's up to 48x, with Wang et al. reporting similar overhead. SharC's overhead is no higher than 14%, whereas Shoal's was up to 42%; SharC exhibited lower overhead than our system in three out of the four benchmarks in common. On the other hand, SharC is unsound in the presence of memory errors, and deals awkwardly with changes in ownership due to its unique pointer restriction (though this can be worked around at some additional performance cost). Isolator has a low overhead (their worst case is a microbenchmark with only 20% slowdown) but it is also unsound in the presence of memory errors and sharing policies cannot change. Hammer et al.'s tool has a slowdown factor between 1.3x and 45x depending on the benchmark.

7. Conclusions

Concurrency and memory errors are hard to debug and they can be exploited by attackers. We presented a dynamic analysis tool for C/C++ programs that can find these errors and prevent attackers from exploiting them. Programmers declare an ownership policy by annotating their code with simple assertions and our tool inserts access checks that detect policy violations at runtime. We proved a modular serializability property that allows local reasoning about program correctness: if a thread executes a sequence of statements on variables it owns, the statements are serializable within a valid execution. We implemented our analysis carefully to ensure this property regardless of what other threads do. We also proved that a check and the access it precedes do not need to be atomic and that many checks can be removed. This allowed us to implement efficient access checks and to use static analysis to remove checks. We evaluated our tool using seven benchmarks. The results show that annotating the benchmarks with ownership assertions required adding or changing only 0.2% of the total number of lines and that the overhead is low. We believe our analysis can be used in practice to find bugs and prevent attacks; it found eight bugs in our benchmark programs.

Acknowledgments

We thank Peter Sewell, Matthew Parkinson, and Tim Harris, who contributed to the development of this work, and Eric Koskinen, Avik Chaudhuri, Nikhil Swamy, Iulian Neamtiu, Polyvios Pratikakis, and Elnatan Reisner who provided useful comments on earlier drafts. Hicks was supported by Microsoft Research as a Visiting Researcher, and in part by NSF grants CCF-0541036 and CNS-0346989.

References

[1] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, May 2008.

[2] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. In *PLDI*, 2008.

[3] Z. Anderson, D. Gay, and M. Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *PLDI*, 2009.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[5] M. Castro, M. Costa, J. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *SOSP*, 2009.

[6] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Detecting atomicity violations via integrated dynamic and static analysis. In *FASE*, 2009.

[7] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low-level programming. In *ESOP*, 2007.

[8] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *PLDI*, 2007.

[9] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109, 2008.

[10] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[11] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.

[12] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[13] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP'09*, 2009.

[14] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, 2008.

[15] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *CAV*, 2009.

[16] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[17] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[18] Phoenix. http://connect.microsoft.com/phoenix.

[19] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *PLDI*, 2006.

[20] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS*, 2009.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[22] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, 2006.

A. Proof of Lemma 4

Definition 11 (Ownership map well-formedness)**.** *Ownership maps are* well-formed*, written* $\vdash\ \omega, \rho$*, iff for all* x, t*:* $\omega(x) = t$ *implies* $\rho(x) \subseteq \{t\}$*; and* $t \in \rho(x)$ *implies either* $\omega(x) = \bot$ *or* $\omega(x) = t$*.*

Lemma 12 (Ownership map well-formedness preservation)**.** *For all* $\alpha, \omega, \rho, \omega', \rho', t, x$ *such that* $\omega, \rho \ \vdash_0\ \alpha \rightsquigarrow \omega', \rho'$ *we have* $\vdash\ \omega, \rho$ *implies* $\vdash\ \omega', \rho'$*.*

Proof. By induction on validation derivations. □

Lemma 4 (Movers, page 6):

Proof. It is sufficient to consider traces $a, b, \cdot$ since lengthier traces can easily be constructed by adding to the beginning or end of the two-element trace. Moreover, it is easy to see that for a valid execution trace, if $var(a) \neq var(b)$, events a and b operate on disjoint parts of the heap and ownership maps, so they can always be safely commuted. Thus we must only consider the cases where $var(a) = var(b)$.

First we consider $op(a) \in \{\mathsf{ownEx}, \mathsf{ownRd}, \mathsf{rd}, \mathsf{wr}\}$, the *right movers*. For $op(a) = \mathsf{ownEx}$, we can show that no valid trace can have $var(a) = var(b)$, so the result follows immediately. For $op(a) = \mathsf{ownRd}$, the only operations by a thread t' with $var(a) = var(b)$ are $\mathsf{ownRd}, \mathsf{relRd}, \mathsf{rd}$. As ownRd places no constraints on the heap, nor does it modify it, given $\sigma \longrightarrow^{\mathsf{ownRd}(t,x)} \sigma \longrightarrow^{b} \sigma'$ we easily have $\sigma \longrightarrow^{b} \sigma' \longrightarrow^{\mathsf{ownRd}(t,x)} \sigma'$. Validity follows essentially because reader-oriented operations are independent—a read or acquirement/release of read ownership by one thread in no way affects the ownership/non-ownership for reading by another thread. The arguments for $op(a) = \mathsf{wr}$ and $op(a) = \mathsf{rd}$ are similar to the arguments for $op(a) = \mathsf{ownEx}$ and $op(a) = \mathsf{ownRd}$, respectively.

Now we consider $op(b) \in \{\mathsf{relEx}, \mathsf{relRd}, \mathsf{rd}, \mathsf{wr}\}$, the *left movers*. For $op(b) = \mathsf{relEx}$, we can show that no valid trace can have $var(a) = var(b)$. This relies on showing that $\omega_1; \rho_1 \ \vdash_0 \mathsf{relEx}(t, x) \rightsquigarrow \omega_2, \rho_2$ implies that $\omega_1(x) = t$, which implies that for any a with $tid(a) \neq t$, we must have $\omega_0(x) = t$ in $\omega_0, \rho_0 \ \vdash_0\ a \rightsquigarrow \omega_1, \rho_1$. By Lemma 12, $\omega_0(x) = t$ implies $\rho_0(x) \subseteq \{t\}$. For $op(b) = \mathsf{relRd}$, we first establish that $op(a) \in \{\mathsf{ownRd}, \mathsf{relRd}, \mathsf{rd}\}$; this again relies on Lemma 12, this time to get $t \in \rho_0(x)$ implies $\omega_0(x) = \bot$ or t. Again, validity follows essentially because these reader-oriented operations operate essentially independently on the various maps, and the heap is not modified. The argument for $op(b) = \mathsf{rd}$ is similar to $op(b) = \mathsf{relRd}$ and the argument for $op(b) = \mathsf{wr}$ is similar to $op(b) = \mathsf{relEx}$.

□