ShArC: Checking Data Sharing Strategies for Multithreaded C

Zachary Ryan Anderson
David Gay
Rob Ennals
Eric Brewer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-25
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-25.html

March 27, 2008

SharC: Checking Data Sharing Strategies for Multithreaded C

Zachary Anderson David Gay[†] Rob Ennals[†] Eric Brewer

University of California, Berkeley
{zra, brewer}@cs.berkeley.edu

[†]Intel Research, Berkeley
{david.e.gay,robert.ennals}@intel.com

Abstract

Unintended or unmediated data sharing is a frequent cause of insidious bugs in multithreaded programs. We present a tool called ShарC (short for *Sharing Checker*) that allows a user to write lightweight annotations to declare how they believe objects are being shared between threads in their program. SharC uses a combination of static and dynamic analyses to check that the program conforms to this specification.

SharC allows any type to have one of five "sharing modes" — private to the current thread, read-only, shared under the control of a specified lock, intentionally racy, or checked dynamically. The dynamic mode uses run-time checking to verify that objects are either read-only, or only accessed by one thread. This allows us to check programs that would be difficult to check with a purely static system. If the user does not give a type an explicit annotation, then SharC uses a static type-qualifier analysis to infer that it is either private or should be checked dynamically.

SharC allows objects to move between different sharing modes at runtime by using reference counting to check that there are no other references to the objects when they change mode.

SharC's baseline dynamic analysis can check any C program, but is slow, and will generate false warnings about intentional data sharing. As the user adds more annotations, false warnings are reduced, and performance improves. We have found in practice that very few annotations are needed to describe all sharing and give reasonable performance. We ran SharC on 6 legacy C programs, summing to over 600k lines of code, and found that a total of only 60 simple annotations were needed to remove all false positives and to reduce performance overhead to only 2–14%.

1. Introduction

The ongoing migration of mainstream processors to many cores accelerates the need for programmers to write concurrent programs. Unfortunately, programmers generally find concurrent programs much more difficult to write than sequential programs. One significant reason for this is that unintended data sharing among threads can make program behavior hard to understand or predict. By *unintended sharing* we mean data shared among threads in ways that the programmer does not expect and that are not part of the design of the program. One key symptom of unintended sharing is a *data race* — when two or more threads access the same memory location without synchronization, and at least one access is a write. However, rather than looking only for data races, we consider most data sharing, even if mediated by locks or signaling, to be an error unless it has been explicitly declared by the programmer. We find in large real programs that requiring sharing to be explicitly declared is not unreasonable.

Unintended or unmediated sharing is often considered a major source of hard-to-find bugs in low-level multithreaded programs. Searching for vulnerabilities caused by race conditions in the US-CERT advisory archive yields hundreds of results [24]. The unintended sharing that causes data races is difficult to diagnose because its occurrence and its effects are highly dependent on the scheduler. This fact may frustrate developers using traditional tools such as a debugger or print statements, which may alter the schedule. Even when unintended sharing is benign, it may indicate places the programmer may need to investigate more carefully.

In this paper, we present SharC, a tool that allows a C programmer to declare the data sharing strategy that a program should follow, and then uses static and dynamic analysis to check that the program conforms to this strategy. Dynamic analysis allows SharC to check any program, including programs that would not be easily checkable with purely static analysis. SharC's static analysis improves performance by avoiding dynamic checks where unintended sharing is impossible.

When using SharC, a programmer uses type qualifiers to declare the *sharing mode* of objects. Sharing modes can declare that an object is thread-private, read-only, protected by a lock, intentionally accessed in a racy way, or checked dynamically. The dynamic sharing mode checks at run time that the object is either read-only, or only accessed by one thread. This allows SharC to check programs that would be difficult to check with a purely static system. The annotation burden is low because SharC makes reasonable and predictable guesses at unannotated types. Further, SharC allows the programmer to change an object's sharing mode using a cast, and uses reference counting to check the safety of such casts dynamically (Section 2).

We show that SharC's mixed static/dynamic checking approach is sound (i.e., detects all data races and checks all sharing mode changes), for a simple C-like language with thread-private and dynamic sharing modes (Section 3). We then describe SharC's implementation, in particular how it selects sharing modes for unannotated types and implements its runtime checks (Section 4).

SharC can be applied to programs incrementally. As more annotations are added, the false positive rate drops and performance improves. We ran SharC on a set of real legacy C programs, which sum to over 600k lines of code. Two of the benchmarks are over 100k lines of code, and some contain kernels that have been finely tuned for performance. Further, we chose benchmarks that use threads in a variety of ways, from threads for serving clients, to hiding I/O and network latency, to improving performance. We found that for this set of programs, the number of annotations required to eliminate false positives and achieve reasonable performance (i.e., 9% runtime and 26% memory overheads) is small (Section 5). These results provide good evidence for the generality and practicality of our approach.

In summary, this paper makes the following novel contributions:

- We specify a lightweight type annotation system that uses a mixture of statically and dynamically checked properties to describe many of the data sharing conventions used in existing multithreaded C programs. We show that in practice, these annotations are not overly burdensome. Indeed we believe that

they are sufficiently simple and clear that they could be viewed as compiler-checked documentation.

- We show that dynamic reference counting makes it possible to safely cast an object between different sharing modes.
- We describe a new dynamic race detection algorithm that uses and enforces the sharing mode annotations supplied by the programmer, and those discovered by a flow-insensitive type qualifier analysis.
- We describe our implementation, and demonstrate its practicality by applying it to several large, real-world, legacy[1] C programs. On these programs we observe lower performance overheads than existing dynamic race detection tools. We believe these overheads are low enough that our analysis could conceivably be left enabled in production systems.

Our implementation also includes an adaptation of the fast concurrent reference counting algorithm found in the work of Levanoni and Petrank [16]. We describe our modifications (Section 4.3), as they may also be useful in memory management schemes for C that rely on reference counting and wish to handle multithreaded programs efficiently.

2. SharC's API

SharC uses a mixture of static and dynamic checking to provide an API that is simple to understand, and yet is complete enough that large real programs can be checked with reasonable runtime efficiency. SharC's first key concept is a set of *sharing mode* annotations that describe the rules that threads must follow when accessing a particular object. The programmer can annotate types with the following sharing mode qualifiers:

- `private`: Indicates that an object is owned by one thread and only accessible by this thread (enforced statically).
- `readonly`: Indicates that an object is readable by any thread but not writeable, with one exception: a `readonly` field in a `private` structure is writeable (enforced statically). This exception makes initialization of `readonly` structure fields practical, while maintaining the read-only property when the structure is accessible from several threads. Unlike C's existing `const` qualifier, `readonly` may be safely cast to another sharing mode that allows writes, as described below.
- `locked`(*lock*): Indicates that an object is protected by a lock and only accessible by whatever thread currently holds the lock (enforced by a runtime check). Here, *lock* is an expression or structure field for the address of a lock, which must be verifiably constant (uses only unmodified locals or `readonly` values) for type-safety reasons.
- `racy`: A mode for objects on which there are benign races (no enforcement required).
- `dynamic`: Indicates that SharC is to enforce at runtime that the object is either read-only, or only accessed by a single thread.

SharC selects a sharing mode for each unannotated type, using a set of simple rules and a whole-program sharing analysis that infers which objects may be accessed by more than one thread. The rules were selected based on our experience using SharC, and help keep the annotation burden low in practice. These rules include a limited form of qualifier polymorphism for structures: unqualified fields inherit the qualifier of their containing structure instance. The result of the sharing analysis is used to add `dynamic` and `private` qualifiers. Unannotated types that refer to objects accessible by more than one thread should be checked for races, so they get the `dynamic` qualifier. All remaining unannotated types are given the `private` qualifier. The qualifiers inferred by this analysis are not trusted: they are statically checked for well-formedness, and by our dynamic analysis. The rules and analysis are described in detail in Section 4.1.

The `dynamic` sharing mode makes SharC practical for large, real-world programs by avoiding the need for complex polymorphic types. However, in real programs, objects often go through a sequence of sharing modes. For example, in a producer-consumer relationship, an object is first private to the producer, then protected by a lock, then private to the consumer. Thus, SharC's second key feature is a *sharing cast* that allows a program to change an object's sharing mode:

```
int readonly *y; ... x = SCAST(int private *, y);
```

SharC enforces the soundness of these casts by nulling-out the pointer being cast, and by using reference-counting to ensure that the pointer being cast is the *only* reference to the underlying object. If we have the only reference to some object, then we can, e.g., safely cast it to or from `private`, since no thread will be able to see the object with the old sharing mode.

SharC can infer where sharing casts are needed to make a program type-check. However, since nulling-out a cast's source may break the program, SharC does not insert these casts automatically, but instead suggests where they should be added. It is then up to the programmer to add the suggested casts to the program if they are safe, or make alternative changes if they are not. Additionally, SharC will emit a warning if a pointer is definitely live after being nulled-out for a cast.

2.1 An Example

Consider a multithreaded program in which a buffer is read in from a file, passed among threads in a pipeline for various stages of processing, and then output to disk, screen, or other medium. This is a common architecture for multimedia software, for example the GStreamer framework [10].

The core function implementing such a pipeline is shown in Figure 1. The `stage` structures are arranged in a list, and there is a thread for each stage. Each thread waits on a condition variable for `sdata` (line 16). When data arrives, the pointer is copied into a local variable (line 17), the original pointer is nulled (line 2), and the previous stage is signaled (line 19). This indicates that the current stage has taken ownership of the data, and that the previous stage is free to use the `sdata` pointer of the current stage for the next chunk of data. Having claimed the data, the current stage processes it somehow (line 21), waits for the next stage to be ready to receive it (line 25), copies the pointer into the next stage (line 27), and then signals that the next stage may claim it (line 28). The process then repeats until all chunks have been processed.

SharC will compile this code as is. However, since the programmer has not added annotations to tell SharC the desired sharing strategy, SharC will assume that all sharing it sees is an error, and will generate an error report. SharC reports two kinds of data sharing. First, it reports sharing of the `sdata` field of the `stage` structures. The following is an example of such a report.

```
read conflict(0x75324464):
   who(2) S->sdata @ pipeline_test.c: 15
   last(1) nextS->sdata @ pipeline_test.c: 27
```

This indicates that thread 2 tried to read from address `0x75324464` through the l-value `S->sdata` on line 15 of the file after thread 1 wrote to the address through l-value `nextS->sdata` on line 27.

[1] By "legacy code" we mean any existing (including multithreaded) C code. We contrast SharC with languages intended for new code(e.g. Cilk [3], and Cyclone [13]).

```
1  // pipeline_test.c
2  typedef struct stage {
3    struct stage *next;
4    cond *cv;
5    mutex *mut;
6    char *locked(mut) sdata;
7    void (*fun)(char private *fdata);
8  } stage_t;
9
10 void *thrFunc(void *d) {
11   stage_t *S = d, *nextS = S->next;
12   char *ldata;
13   while (notDone) {
14     mutexLock(S->mut);
15     while (S->sdata == NULL)
16       condWait(S->cv,S->mut);
17     ldata = SCAST(char private *, S->sdata);
18     S->sdata = NULL;
19     condSignal(S->cv);
20     mutexUnlock(S->mut);
21     S->fun(ldata);
22     if (nextS) {
23       mutexLock(nextS->mut);
24       while (nextS->sdata)
25         condWait(nextS->cv,nextS->mut);
26       nextS->sdata =
27         SCAST(char locked(nextS->mut) *, ldata);
28       condSignal(nextS->cv);
29       mutexUnlock(nextS->mut);
30     }
31   }
32   return NULL;
33 }
```

Figure 1. A simple multithreaded pipelining scheme as might be used in multimedia software. Items in bold are the additions for SharC.

Without knowledge of the desired sharing strategy, SharC assumes that this sharing is an error.

This error report was generated by SharC's dynamic checker. SharC was not able to prove statically that the `sdata` field was private, so it inferred the `dynamic` sharing mode for `sdata`, and monitored it at runtime for races (two threads have accessed the same location, with at least one access being a write).

As a human reading the code, it is clear that the programmer intended the `sdata` field to be shared between threads, and to be protected by the `mut` lock. We can declare this policy to SharC by adding a `locked` annotation to line 6. Now, rather than checking that `sdata` is not accessed by more than one thread, SharC will instead check that the referenced lock is held whenever `sdata` is accessed.

SharC will also report sharing of memory *pointed to* by the `sdata` field. Here is an example:

```
write conflict(0x75324544):
  who(2) *(fdata + i) @ pipeline_test.c: 52
  last(3) *(fdata + i) @ pipeline_test.c: 62
```

Lines 52 and 62 are not shown in the figure, but are both in functions that can be pointed to by the `fun` field of the `stage` structure. In these functions, `fdata` is equal to the `ldata` pointer that was passed as the argument to `fun`. The error message indicates that thread 2 tried to write to address `0x75324544` through the l-value `*(fdata+i)` on line 52 after thread 3 had read from the same address through the l-value `*(fdata+i)` on line 62.

Here, SharC is not aware that ownership of the buffer is being transferred between the threads, and so believes that the buffer is being shared illegally. The user can tell SharC what is going on by adding a `private` annotation to the `fdata` argument of `fun` on

```
1  // pipeline_test.c
2  typedef struct stage(q) {
3    struct stage pdynamic *q next;
4    cond racy *q cv;
5    mutex racy *readonly mut;
6    char locked(mut) *locked(mut) sdata;
7    void (*q fun)(char private *private fdata);
8  } stage_t;
9
10 void dynamic*private thrFunc(void dynamic *private d){
11   stage_t dynamic *private S = d;
12   stage_t dynamic *private nextS = S->next;
13   char private *private ldata;
14   ...
15 }
```

Figure 2. The `stage` structure, with the annotations inferred by SharC shown un-bolded. The qualifier polymorphism in structures is shown through the use of the qualifier variable `q`.

line 7. This will cause SharC to infer that `ldata` is `private` rather than `dynamic`, but type checking will fail at lines 17 and 27 due to the assignment of a (`char locked(mut)*`) to and from (`char private*`). To fix this, SharC suggests the addition of the sharing casts (SCAST(...)) shown in bold on lines 17 and 27. As discussed above, these sharing casts will null-out the cast value (`S->sdata` or `ldata`) and check that the reference count for the object is one. For line 17 this ensures the object referenced by `S->sdata` is not accessible by inconsistently qualified types (`locked(...)` and `private`). These two annotations and two casts are sufficient to describe this simple program's sharing strategy, and allow it to run without SharC reporting any errors.

Figure 2 shows the sharing modes selected by SharC for the `stage` struct, and the first few lines of the `thrFunc` function. The `next`, `cv` and `fun` fields inherit the structure's qualifier `q`. The internals of pthread's lock and condition variable types (`mutex`, `cond`) have data races by their very nature, so they have the `racy` qualifier. The `mut` field must be `readonly` for type-safety reasons, as the type of `sdata` depends on it. The object referenced by `sdata` has "inherited" its pointer's sharing mode. The object referenced by `next` has not been annotated, so must be given the `dynamic` mode since the structure's qualifier can't be similarly "inherited" for referent types for soundness reasons. SharC's sharing analysis infers that the object passed to `thrFunc` is accessible from several threads, so `d`, `S` and `nextS` must be pointers to `dynamic`.

At runtime for the annotated program, SharC will check that:

1. When the `sdata` field of a `stage` structure is accessed, the `mut` mutex of that structure is held.
2. When the non-`private` pointer `S->sdata` is cast to `private` on line 17, no other references to that memory exist.
3. When the `private` pointer `ldata` is cast to a non-`private` type on line 27, no other references to that memory exist.
4. There are no races on objects inferred to be in `dynamic` mode.

3. The Formal Model

In this section we present a formal model for SharC. We reduce the set of sharing modes to just `private` and `dynamic`, omitting `racy`, `locked`, and `readonly`. We also use a simplified version of the C language. Our goal is to express the essence of SharC, without obscuring it with these additional features. The formalism is readily extendable to include `locked`, `readonly`, and `racy`.

Core Type	σ	::=	int | ref τ
Sharing Mode	m	::=	`dynamic` | `private`
Type	τ	::=	$m\ \sigma$ | thrd
Program	P	::=	$\tau\ x$ | $f()\{\tau_1\ x_1 \ldots \tau_n\ x_n; s\}$ | $P; P$
L-expression	ℓ	::=	x | $*\,x$ | $\boldsymbol{a}$
Expression	e	::=	ℓ | $\text{scast}_\tau\ x$ | n | null | new_τ
Statement	s	::=	$s_1; s_2$ | spawn $f()$
		|	$\ell := e$ [when $\omega_1(\ell_1), \ldots, \omega_n(\ell_n)$]
		|	**skip** | **done** | **fail**
Predicate	ω	::=	*chkread* | *chkwrite* | *oneref*

$f, x \in$ Identifiers $\quad a, n \in$ Integer constants

Figure 3. The grammar for a simple concurrent language with global and local variables, and qualifiers for describing sharing. **done**, **skip**, **fail** and runtime addresses ($\boldsymbol{a}$) are only used in the operational semantics.

3.1 Language

Figure 3 shows the grammar for the language. A type (τ) includes a `private` or `dynamic` sharing mode. Programs (P) consist of a list of declarations of thread definitions and global variables. Thread definitions (f) have no arguments or results and declare local variables. We assume that all identifiers are distinct. The thread body is a sequence of statements (s) that spawn threads and perform assignments. Control flow has no effect on our type system or runtime checks, so it is omitted from the language. L-values (ℓ) include global and local variables, and dereferences of local variables (this restriction is enforced in the type system). Assignments can assign l-values, constants (n, null) and newly allocated memory cells, and perform casts that change the sharing mode of a referenced cell (scast x). Note that casts implicitly null-out their source, to eliminate the reference with the old type. Runtime checks (added during type checking) are represented by guards (when $\omega_1, \ldots$) on assignments. The runtime checks consist of assertions that it is safe to access memory cells (*chkread*, *chkwrite*), and assertions that there is only one reference to a cell (*oneref*).

3.2 Static Semantics

The static type-checker (Figure 4) checks that programs are well-typed and inserts runtime checks for casts and accesses to objects with `dynamic` sharing mode. The rules for checking programs ensure that global declarations use the `dynamic` sharing mode (GLOBAL) and that no type has a `dynamic` reference to a `private` type (REF CTOR). The thrd type is used to identify threads in Γ and cannot appear in user-declared types. The five assignment rules (*-ASSIGN) check the types being assigned and rely on the R and W functions to compute runtime checks for accesses to objects in `dynamic` mode.

Sharing casts (CAST-ASSIGN) allow, e.g., the conversion of a ref (`dynamic` int) to a ref (`private` int). To guarantee soundness, this sharing cast is preceded by a *oneref* check that ensures that the converted reference is (at runtime) the sole reference to the `dynamic` int. It is not possible to convert types that differ further down, e.g., you cannot cast from ref (`dynamic` ref (`dynamic` int)) to ref (`private` ref (`private` int)) as the existence of a single reference to (`dynamic` ref (`dynamic` int)) does not guarantee that there are not more references to the underlying int.

To simplify the proof of correctness, we require that x is `private`, and hence a local variable, in the CAST-ASSIGN and DEREF rules. It is easy to rewrite a program with extra local variables to meet this requirement. The remaining rules are straightforward and self explanatory.

$\Gamma \vdash P \Rightarrow P'$ In environment Γ program P compiles to P', which is identical to P except for added runtime checks.

$$\text{(GLOBAL)}\ \frac{\vdash \texttt{dynamic ref}\ \tau \qquad \Gamma[x:\tau] \vdash P \Rightarrow P'}{\Gamma \vdash \tau\ x; P \Rightarrow \tau\ x; P'}$$

$$\text{(THREAD)}\ \frac{\vdash \tau_i \qquad \Gamma[x_1:\tau_a, \ldots, x_n:\tau_n] \vdash s \Rightarrow s' \qquad \Gamma[f:\text{thrd}] \vdash P \Rightarrow P'}{\Gamma \vdash f()\{\tau_1\ x_1 \ldots \tau_n\ x_n; s\}; P \Rightarrow f()\{\tau_1\ x_1 \ldots \tau_n\ x_n; s'\}; P'}$$

$\vdash \tau$ Type τ has no `dynamic` references to `private` types

$$\text{(INT CTOR)}\ \frac{}{\vdash m\ \text{int}} \qquad \text{(REF CTOR)}\ \frac{\vdash m'\sigma \qquad m = m' \vee m = \texttt{private}}{\vdash m\ \text{ref}\ (m'\sigma)}$$

$\Gamma \vdash \ell : \tau$ In environment Γ, ℓ is a well-typed lvalue with type τ.

$$\text{(NAME)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{(DEREF)}\ \frac{\Gamma(x) = \texttt{private ref}\ \tau}{\Gamma \vdash *x : \tau}$$

$\Gamma \vdash s \Rightarrow s'$ In environment Γ statement s compiles to s', which is identical to s except for added runtime checks.

$$\text{(SEQ)}\ \frac{\Gamma \vdash s_1 \Rightarrow s_1' \qquad \Gamma \vdash s_2 \Rightarrow s_2'}{\Gamma \vdash s_1; s_2 \Rightarrow s_1'; s_2'} \qquad \text{(SPAWN)}\ \frac{\Gamma(f) = \text{thrd}}{\Gamma \vdash \text{spawn}\ f() \Rightarrow \text{spawn}\ f()}$$

$$\text{(CONSTANT-ASSIGN)}\ \frac{\Gamma \vdash \ell : m\ \text{int}}{\Gamma \vdash \ell := n \Rightarrow \ell := n\ \text{when}\ \text{W}(\ell, m)}$$

$$\text{(NULL-ASSIGN)}\ \frac{\Gamma \vdash \ell : m\ \text{ref}\ \tau}{\Gamma \vdash \ell := \text{null} \Rightarrow \ell := \text{null when W}(\ell, m)}$$

$$\text{(NEW-ASSIGN)}\ \frac{\Gamma \vdash \ell : m\ \text{ref}\ \tau}{\Gamma \vdash \ell := \text{new}_\tau \Rightarrow \ell := \text{new}_\tau\ \text{when}\ \text{W}(\ell, m)}$$

$$\text{(ASSIGN)}\ \frac{\Gamma \vdash \ell_1 : m_1\ \sigma \qquad \Gamma \vdash \ell_2 : m_2\ \sigma}{\Gamma \vdash \ell_1 := \ell_2 \Rightarrow \ell_1 := \ell_2\ \text{when}\ \text{W}(\ell_1, m_1), \text{R}(\ell_2, m_2)}$$

$$\text{(CAST-ASSIGN)}\ \frac{\begin{array}{c}\Gamma \vdash \ell : m\ \text{ref}\ (m_1\ \sigma) \\ \Gamma(x) = \texttt{private ref}\ (m_2\ \sigma) \qquad \tau = m_1\ \sigma\end{array}}{\Gamma \vdash \ell := \text{scast}_\tau\ x \Rightarrow \ell := \text{scast}_\tau\ x\ \text{when}\ \textit{oneref}(*x), \text{W}(\ell_1, m)}$$

Types with dynamic sharing mode need runtime checks

$$\text{R}(\ell, \texttt{dynamic}) = \textit{chkread}(\ell) \qquad \text{R}(\ell, \texttt{private}) = \epsilon$$
$$\text{W}(\ell, \texttt{dynamic}) = \textit{chkwrite}(\ell) \qquad \text{W}(\ell, \texttt{private}) = \epsilon$$

Figure 4. Typing judgments.

$$S ::= [\cdot]_S \mid S; s \mid l := e \text{ when } \omega_1([\cdot]_L), \omega_2(\ell_2), \ldots, \omega_n(\ell_n) \mid [\cdot]_L := e \mid a := [\cdot]_L$$

$$\begin{array}{rcl@{\qquad}rcl}
M, E : x & \xrightarrow{\ell} & E(x) & M, E, id : \textbf{skip}; s & \xrightarrow{s} & M, \circ, s \\
M, E : *x & \xrightarrow{\ell} & M(E(x)).\text{value} & M, E, id : \text{spawn } f() & \xrightarrow{s} & M, f, \textbf{skip} \\
M, E, id : a_1 := a_2 & \xrightarrow{s} & M[a_1 \xrightarrow{v} M_v(a_2)], \circ, \textbf{skip} & M, E, id : a := \text{null} & \xrightarrow{s} & M[a \xrightarrow{v} 0], \circ, \textbf{skip} \\
M, E, id : a := \text{new}_\tau & \xrightarrow{s} & extend(M[a \xrightarrow{v} m+1], id, \tau), \circ, \textbf{skip} & M, E, id : a := n & \xrightarrow{s} & M[a \xrightarrow{v} n], \circ, \textbf{skip}
\end{array}$$

$$M, E, id : a_1 := \text{scast}_\tau\ a_2 \xrightarrow{s} M[a_1 \xrightarrow{v} v_2, a_2 \xrightarrow{v} 0, v_2 \xrightarrow{\tau} \tau, v_2 \xrightarrow{o} id, v_2 \xrightarrow{R} \emptyset, v_2 \xrightarrow{W} \emptyset], \circ, \textbf{skip where } v_2 = M(a_2).\text{value}$$

$$M, E, id : \ell := e \text{ when } \omega_1(a), \omega_2(\ell_2) \ldots, \omega_n(\ell_n) \xrightarrow{s} M', \circ, \ell := e \text{ when } \omega_2(\ell_2), \ldots, \omega_n(\ell_n) \textbf{ if } M, id \models \omega_1(a) \Rightarrow M'$$

$$M, E, id : \ell := e \text{ when } \omega_1(a), \omega_2(\ell_2) \ldots, \omega_n(\ell_n) \xrightarrow{s} M, \circ, \textbf{fail if } M, id \not\models \omega_1(a)$$

$$\frac{M, E, id : s \xrightarrow{s} M', f, s'}{M, E, id : S[s]_S \to M', f, S[s']_S} \qquad \frac{M, E : \ell \xrightarrow{\ell} \ell' \qquad \ell' \neq 0}{M, E, id : S[\ell]_L \to M, \circ, S[\ell']_L} \qquad \frac{M, E : \ell \xrightarrow{\ell} \ell' \qquad \ell' = 0}{M, E, id : S[\ell]_L \to M, \circ, \textbf{fail}}$$

$$\frac{M, E, i : s_i \to M', \circ, s_i'}{M, \ldots, (E_i, s_i), \ldots \to M', \ldots, (E_i', s_i'), \ldots} \qquad \frac{M' = threadexit(M, E_i, i)}{M, \ldots, (E_i, \textbf{skip}), \ldots \to M', \ldots, (E_i, \textbf{done}), \ldots}$$

$$\frac{M, E, i : s_i \to M', f, s_i' \qquad f()\{\tau_1 x_1 \ldots \tau_n x_n; s\} \in P \qquad E = G[x_1 \to m+1, \ldots, x_n \to m+n] \qquad M'' = extend(M', n+1, \tau_1, \ldots, \tau_n)}{M, \ldots, (E_i, s_i), \ldots, (E_n, s_n) \to M'', \ldots, (E_i', s_i'), \ldots, (E_n, s_n), (E, s)}$$

$$m = \max(\text{dom}(M))$$

$$extend(M, id, \tau_1, \ldots, \tau_k) = M[\ldots, (m+i) \xrightarrow{\tau} \tau_i, (m+i) \xrightarrow{v} 0, (m+i) \xrightarrow{o} id, (m+i) \xrightarrow{R} \emptyset, (m+i) \xrightarrow{W} \emptyset, \ldots]$$

$$threadexit(M, E, id) = M' \text{ where } \begin{cases} M_v'(a) = 0 \textbf{ if } \exists x \in \text{dom}(E) - \text{dom}(G).E(x) = a \\ M_v'(a) = M_v(a) \textbf{ otherwise} \\ M_\tau'(a) = M_\tau(a), M_R'(a) = M_R(a) - \{id\}, M_W'(a) = M_W(a) - \{id\} \end{cases}$$

Figure 6. Parallel operational semantics.

$$\frac{M_W(a) - \{id\} = \emptyset}{M, id \models chkread(a) \Rightarrow M[a \xrightarrow{R} M_R(a) \cup \{id\}]}$$

$$\frac{M_R(a) - \{id\} = \emptyset \qquad M_W(a) - \{id\} = \emptyset}{M, id \models chkwrite(a) \Rightarrow M[a \xrightarrow{W} M_W(a) \cup \{id\}]}$$

$$\frac{|\{b | M(b).\text{value} = a \wedge M(b).\text{type} = m \text{ ref } \tau\}| = 1}{M, id \models oneref(a) \Rightarrow M}$$

Figure 5. Semantics for performing runtime checks. Checks are executed in one big step once their argument is known.

3.3 Dynamic Semantics

In Figures 5 and 6 we give a small-step operational semantics for our simple language. This semantics gives a formal model for the way that SharC checks accesses to objects in `dynamic` mode, and checks casts between sharing modes.

The runtime environment consists of three mappings and an integer thread identifier:

- **Memory** $M : \mathbb{N} \to \mathbb{Z} \times \tau \times \mathbb{N} \times \mathscr{P}(\mathbb{N}) \times \mathscr{P}(\mathbb{N})$

  M maps a cell's address to its value, type, owner, and sets of readers and writers (sets of thread identifiers). For cells of reference type, the value is the address of the referenced cell (0 is always an invalid address). For cells in `private` mode, the owner is the thread that can access the cell. We use the notations M_v, M_τ, M_o, M_R and M_W to denote the projection of M to (respectively) it's value, type, owner, readers and writers.

  Correspondingly, $M[a \xrightarrow{v} n]$ is the memory that differs from M only by the fact that $M[a \xrightarrow{v} n]_v(a) = n$ (and so on for the other elements of a cell). At program start M maps the global variables to zeroes of the appropriate type, with empty reader and writer sets, and owner = 0 (no owner).

  Note that SharC's implementation does not need to track types or owners at runtime (the type and owner elements are never read by the operational semantics).

- **Globals** $G : var \to \mathbb{N}$ gives the address of each global variable in M. G never changes, so we use it directly when needed.
- **Environment** $E : var \to \mathbb{N}$ gives the addresses in M of the variables of a particular thread (locals and globals).
- **Thread** $id : \mathbb{N}$ is a positive natural number identifying a thread.

There are three kinds of transition rules and a runtime judgment:

- $M, E : \ell \xrightarrow{\ell} a$ — In the given environment the l-value ℓ evaluates to the address a where the l-value is stored.
- $M, E, id : s \xrightarrow{s} M', f, s'$ — In the given environment, the execution of statement s results in new memory M' and proceeds to statement s'. If $f \neq \circ$, thread f should be spawned.
- $M, (E_1, s_1), \ldots, (E_n, s_n) \to M, (E_1, s_1'), \ldots, (E_m, s_m')$ — A system with main memory M and n threads with environments E_i currently executing statement s_i transitions to a new state with main memory M' and new thread state. Each step corresponds to a change in a single thread i, so $\forall j \neq i.s_j = s_j'$. The choice of which thread advances at any given point is non-deterministic. Terminated threads are left in the list with

s_k = **done** and at most a single thread is spawned at each step, so $m = n \vee m = n + 1$.

- $M, id \models \omega(a) \Rightarrow M'$ — Given memory state M, runtime check ω is satisfied for the cell at address a with new reads and writes by id recorded in M'.

  The *chkread* and *chkwrite* checks update readers and writers, as the check and update must be done atomically, but the subsequent reads and writes can be performed independently assuming the runtime check succeeds.

Most of the semantic rules are straightforward. The do-nothing **skip** statement is used internally to simplify the semantics. Runtime checks (when) are executed left-to-right before the assignment that they guard. If a runtime check fails, or if a null pointer is dereferenced, the thread transitions to the special **fail** statement, leaving the thread blocked. The *chkread* and *chkwrite* runtime checks (Figure 5) enforce the n-readers-or-1-writer restriction for `dynamic` cells. The *extend* function extends the memory map with n new memory cells of the specified type, initialized to zero. Finally, *threadexit* updates the memory after a thread exits: the thread is removed from all readers and writers sets, and the cells containing the thread's locals are set to zero.

The most interesting part of the semantics is the rule for scast. First, as enforced by the static semantics, the *oneref* predicate is used to check that there is a single reference to the object referenced by local variable x. This guarantees that it is safe to change the type of $*x$ as specified by the cast. Second, x itself must be set to null, as there would otherwise be two ways of accessing the same memory cell with inconsistent types. Finally, the modified memory cell has its reader and writer sets cleared: after a cast, past accesses by other threads no longer constitute unintended sharing since the user has explicitly changed the cell's sharing mode, and verified that other threads can no longer access it.

The *oneref* check is specified using heap inspection for simplicity of exposition, while the SharC implementation uses reference counting. This does not affect our semantics as our simple language does not support recursive types.

3.4 Soundness

We have proved that these semantics guarantee that the declared sharing strategy is respected:

- `private` cells are only accessed by the thread that owns them
- no two threads have a race on a `dynamic` cell, i.e., access the same `dynamic` cell with at least one access being a write, unless there has been an intervening sharing cast

The proof is based on showing that at all times, all threads are well-typed, well-checked (necessary runtime checks hold or will be performed), and *consistent* with the memory.

Definition 1. M is *consistent* with the environment E of thread id if types and owners are consistent between E, M and the static semantics, and when one cell refers to another within M. Furthermore, the readers and writers sets in M must have legal values. Formally, for all addresses a with non-zero value $b = M_v(a)$, and all variables x with declared type τ_x:

- $M_\tau(E(x)) = \tau_x$ (variable types are preserved)
- $x \notin \text{dom}(G) \Rightarrow M_o(E(x)) = id$ (local variables are owned by their thread)
- $M_\tau(a) = m \text{ ref } \tau \Rightarrow b \neq E(x)$ (variables are not addressable)
- $M_\tau(a) = m \text{ ref } \tau \Rightarrow M_\tau(b) = \tau$ (types are consistent)
- $M_\tau(a) = \texttt{private ref (private } \sigma) \Rightarrow M_o(a) = M_o(b)$ (owners are consistent)
- $|M_W(a)| \leq 1$ (no more than one writer)
- $M_W(a) \neq \emptyset \Rightarrow M_R(a) \subseteq M_W(a)$ (no other readers than the writer)

Proof Outline: The proof proceeds by induction over steps of the operational semantics, showing that all steps preserve well-typedness, well-checkedness and consistency. From this, it immediately follows that private accesses are safe, i.e., when thread id reads or writes `private` cell a, a's owner is id. Similarly, well-checkedness implies that dynamic accesses are safe, i.e., when thread id reads (respectively, writes) `dynamic` cell a, *chkread* (a) (respectively, *chkwrite* (a)) holds. Therefore no races occur on `dynamic` cells. The full proof is found in Appendix A.

4. Implementation

The input to the SharC compiler is a partially annotated C program. The SharC compiler first infers the missing annotations (Section 4.1). SharC then type-checks and instruments with runtime-checks the now-complete program. This augmented code is then passed to a normal C compiler and linked with SharC's runtime library.

At runtime, SharC verifies correct use of `dynamic` and `locked` locations, and checks that sharing casts are only applied to objects to which there is only one reference (Section 4.2).

SharC assumes that its input is a type- and memory-safe C program. Furthermore, sharing casts that change qualifiers of (`void *`) types are forbidden. The programmer must cast the (`void *`) pointer to a concrete type before the sharing change so that SharC can check that the change is legal. To ensure full soundness, SharC would need to be combined with a system such as CCured [18], or Deputy [5] that checks such (`void *`) casts and guarantees type- and memory-safety.

4.1 Sharing Analysis

The main purpose of SharC's sharing analysis is to determine which data might be shared across threads and needs the `dynamic` mode, with the remaining data being `private`. Additionally, however, to reduce the annotation burden, SharC will infer other sharing modes by following these simple rules:

- A field or variable used in a `locked` qualifier must be `readonly`, to preserve soundness.
- Type definitions can specify that they are inherently racy. This is used, e.g., for pthread's `mutex` and `cond` types.
- SharC provides a simple form of qualifier polymorphism for structs. If the outermost qualifier on a structure field is not specified, it is inferred to be the same as the qualifier on the *instance* of the structure. This is sound, since structure fields occupy the same memory that is described by the instance. As a consequence, SharC does not allow the outermost annotation of a field to be `private`: within a `private` struct, such a field is already private, while accesses to a `private` field within a non-`private` struct would be unsound.
- Outside of structure definitions, if the target type of a pointer is unannotated, then it is assumed to be the type of the pointer. For instance (`int * dynamic`) becomes (`int dynamic * dynamic`), but (`int dynamic * private`) remains as is. Inside of a structure definition, unannotated pointer target types are given the `dynamic` mode.
- An array is treated like a single object of the array's base type.

We have found that these rules expedite the application of SharC to real C programs.

After these rules have been applied, the sharing analysis makes all remaining unannotated types either `private` or `dynamic`. Be-

cause accesses to `dynamic` objects are checked at runtime to detect data races, SharC attempts to minimize the number of objects inferred to be `dynamic`. First, we describe how the `dynamic` qualifier flows for assignments and function calls. Second, we describe how the analysis is seeded by a set objects that are inherently shared among threads. Taken together, this is sufficient to identify all the potentially shared objects that need the `dynamic` qualifier.

For assignments, we follow CQual's[9] flow-insensitive type qualifier rules, with changes to account for qualifier polymorphism in structures. To avoid overaggressive propagation of the `dynamic` qualifier, we only infer that it flows from formals to actuals in the following case: if a formal is stored in a `dynamic` location, or has a `dynamic` location stored in it, then the `dynamic` qualifier will flow to the actual at the call site. This is equivalent to adding a second kind of `dynamic` qualifier, which we internally refer to as *dynamic_in*, which accepts both `private` and `dynamic` objects. Users of our system never see or write this qualifier.

Next, we must find the shared objects with which to seed the analysis. First, we observe that for an object to be shared, it must be read or written in a function spawned as a thread. The locations available to a function spawned as a thread are the following:

- locals — These can only be shared if their addresses are written into another shared location, so locals are not seeds.
- formals — The argument to the thread function is an object passed by another thread, so is inherently shared and seeds the analysis.
- globals — All globals touched by thread functions might be shared, and so are seeds for the analysis.

It is an error if any of these inherently shared objects have been annotated as `private`.

In order to identify all globals that might be touched by threads, we construct control flow graphs rooted at the functions that are spawned as threads. We handle function pointers by assuming that they may alias any function in the program of the appropriate type. This is sound under our type and memory safety assumption.

4.2 Runtime Checks

At runtime, SharC tracks held locks, reference counts, and reader, writer sets for `dynamic` locations. This information is then inspected in a straightforward way by the various runtime checks.

4.2.1 Tracking Reader and Writer Sets

For every 16 bytes of memory SharC maintains an extra n bytes that record how each thread has accessed those 16 bytes. We can support up to $8n - 1$ threads when n extra bytes are used for record keeping. For the applications we have evaluated with SharC, setting $n = 1$ has been sufficient. Accesses and updates to the bits are made atomic, as required by the *chkread* and *chkwrite* checks, through use of the `cmpxchg` instruction available on the x86 architecture. These extra bytes encode the reader and writer sets as follows. If the low (0-th) bit is set, this indicates that a single thread is reading and writing to the location. If the n-th bit is set, this indicates that the n-th thread is reading from the location, and writing to it if the 0-th bit is also set. This encoding of reader, writer sets does not scale well to larger numbers of threads. In the future, we plan to explore alternative, more efficient encodings.

When heap memory is deallocated with `free()`, it is no longer considered to be accessed by any thread, and all of its bits are cleared. When a thread ends, the bits recording its accesses are cleared: SharC does not consider it a race for two threads to access the same location if their execution does not overlap. The clearing operation is made efficient by logging the addresses of all of a thread's reads and writes on its first accesses to those addresses.

```
1 void *scast(void *src, void **slot) {
2   *slot = NULL;
3   if (refcount(src) > 1)
4     error();
5   return src;
6 }
```

Figure 7. The procedure for checking a sharing cast.

4.2.2 Tracking Held Locks

When a lock is acquired, the address of the lock is stored in a thread private log. When a thread accesses an object in the locked sharing mode, a runtime check is added that ensures the required lock is in the log. When the lock is released, the address of the lock is removed from the log.

4.2.3 Checking Sharing Casts

The procedure for checking a sharing cast is given in Figure 7. When a programmer adds an explicit sharing cast, e.g. `x = SCAST(t, y)`, SharC transforms it into `x = scast(y, &y)` after determining that the cast is legal. The address of the reference is needed so that the reference can be nulled out. Then, if there is more than one reference to the object being casted, an error is signaled. Finally, the reference is returned. In the example above, if `y` were still live after the cast, SharC would issue a warning. This warning lets the programmer know that the refernce will be null after the cast. The procedure for determining reference counts is described in section 4.3.

4.3 Maintaining Reference Counts

SharC builds upon the authors' prior work [12] in reference counting C. Applying this work directly in SharC implies atomically updating reference counts for all pointer writes. The resulting overhead is unacceptable on current hardware, even on x86 family processors that support atomic increment and decrement instructions. To reduce this overhead, SharC performs a straightforward whole-program, flow-insensitive, type-qualifier-like analysis to detect which locations might be subject to a sharing cast. Only pointers to these locations need reference count updates. However, even with this optimization, the runtime overhead is still too high (over 60% in many cases). To reduce this overhead, we adapted Levanoni and Petrank's high performance concurrent reference counting algorithm [16] (designed for garbage collection) for use with SharC.

4.3.1 Overview

In Levanoni and Petrank's algorithm, each thread keeps a local unsynchronized log of the references that have had their values overwritten by that thread. To keep the log small, an entry is only added the first time a location is overwritten since the last collection. A *dirty* bit associated with each location is used to record whether a mutator has previously overwritten that location. To bring all these logs together, a dedicated coordinator thread periodically stops all the mutators, grabs the logs they kept of their mutations, clears the dirty bits, sets the threads running again, and then computes updated reference counts by processing the merged logs. To compute the updated reference counts, the coordinator walks through the grabbed logs, and for each entry, decrements the overwritten references, and increments the reference counts for the references currently in the reference cells. It is necessary to take into account mutations since the threads were restarted. For these, if a reference cell is dirty when the coordinator wants to increment the reference count for its current value, it instead finds the recently overwritten reference value in the live update logs, and increments the reference count for that reference. Although this does not reflect the *current* reference counts, it reflects the reference counts that

```
1  void update(void **slot, void *new) {
2    void *old = *slot;
3    choosing[tid] = 1;
4    dummyCall();
5    rcidxs[tid] = rcidx;
6    choosing[tid] = 0;
7    if (notDirty(rcidxs[tid],slot)) {
8      *log[rcidxs[tid]][tid]++ = (slot,old);
9      markDirty(rcidxs[tid],slot);
10   }
11   rcidxs[tid] = -1;
12   *slot = new;
13 }
```

Figure 8. The procedure for updating a reference.

were correct at that snapshotted time. Levanoni and Petrank also have another algorithm that stops threads one by one, rather than all at once, but which is more complicated to implement.

We have adapted Levanoni and Petrank's simpler algorithm to avoid the need to stop all threads while the coordinator grabs the buffers and clears the dirty bits. In our algorithm, there are two sets of logs and two sets of dirty bits. When the collector thread wants to obtain an accurate snapshot view of the reference counts, it makes all the threads switch to their other update log and the other set of dirty bits (which the coordinator has just cleared). A simple non-blocking algorithm is used to ensure that all threads switch over together. The coordinating thread need only pause as long as it takes the mutator threads to complete any outstanding writes to their logs.

4.3.2 Reference Update

The procedure for updating a reference is given in Figure 8. The goal of this procedure, aside from updating the thread-private log, is to ensure that any thread trying to calculate reference counts knows which set of logs the mutating threads are using. First, the old reference is recorded before the dirty bit is inspected. This is so that a race on the dirty bit will at worst result in a duplicate log entry. Next, we set a flag that indicates we are recording which set of logs this update will use. The reasoning here is similar to that in Lamport's well-known bakery algorithm for mutual exclusion. The coordinating thread may not begin computing while there are threads deciding which set of logs to use since they might decide to use the old set.

The next step is to record the set of logs being used for the update by loading the global variable `rcidx` and storing it into an array that records the logs currently being used by each thread. In the absense of sequential consistency, it is possible that a processor may reorder the load of `rcidx` before the store to `choosing[tid]`. This is a problem since it could result in a thread performing an update using the old set of logs.

In our algorithm we rely only on the guarantees specified by Intel for the x86 architecture[15]. This says that, (1) loads are not reordered with other loads and stores are not reordered with other stores, (2) stores are not reordered with older loads, *but* (3) loads **may** be reordered with older stores to different locations. Fortunately, we can work around point 3 by inserting additional loads and stores to the same location, preventing undesired reordering. The call on line 4 is to a function that does nothing but return. In pushing the return address onto the stack and then popping it off, it performs the needed additional loads and stores, and prevents the processor from reordering the read of `rcidx` before the store to `choosing[tid]`.

After choosing a set of logs, if the reference cell is not dirty, the old value is logged, and the cell is marked dirty. The thread's entry

```
1  int existsOldOrChoosing(int idx) {
2    int i;
3    for (i = 0; i < MAXTID; i++)
4      if (choosing[i] OR rcidxs[i] == idx)
5        return 1;
6    return 0;
7  }
8
9  void updateRefCounts(void *p, int sz) {
10   int oldidx, cnt = 0;
11   void *end = p + sz;
12   Log ← ∅; Undet ← ∅;
13   mutexLock(refcntLock);
14
15   oldidx = rcidx;
16   rcidx = rcidx ? 0 : 1;
17   while (existsOldOrChoosing(oldidx))
18     yield();
19
20   foreach tid ∈ Threads
21     Log ← Log ∪ log[oldidx][tid];
22   foreach (slot,old) ∈ Log
23     void *new = *slot;
24     refcounts[old]--;
25     if (notDirty(rcidx,slot)) refcounts[new]++;
26     else Undet ← Undet + slot;
27   Log ← ∅;
28   foreach tid ∈ Threads
29     Log ← Log ∪ log[rcidx][tid];
30   foreach (slot,old) ∈ Log
31     if (slot ∈ Undet)
32       refcounts[old]++;
33
34   while (p < end) cnt += refcounts[p++];
35   mutexUnlock(refcntLock);
36 }
```

Figure 9. The procedure for calculating reference counts.

in `rcidxs` is set to -1 to indicate that it is no longer using any set of logs, and finally the update is performed.

4.3.3 Reference Count Calculation

The procedure for calculating a snapshot of the reference counts is given in Figure 9. A mutex is acquired so that only one thread may be acting as coordinator at a time. Next, the old logs are swapped out, and then we wait for each of the mutators to start using the new logs. Starvation is impossible since the store to `rcidx` will eventually be visible to all threads. Next, the old logs are merged, and for each of the entries, the reference count for the old value is decremented. If the reference cell has not been updated according to the new logs, the reference count for the new value is incremented. If the reference cell has been updated according to the new logs, we add the cell to a set of "Undetermined" cells. Then, we merge the new logs, and for each entry, if the reference cell for the new entry is in the Undetermined set, the reference count for the old value is incremented. The reference count for the new value will be underestimated, but none of these underestimates will be of any consequence: before calculating a reference count, the reference in question is copied into a local variable and then nulled out. If, during the reference count, another thread is updating a reference cell with the same location we are reference counting, then there must have been more than one reference to the location to begin with. So, we will sometimes underestimate the reference count by one, but only when it would have been three or larger. The bad cast will be detected whether or not there is an underestimate.

4.4 The C Library

In applying SharC to real C programs, it is necessary to handle some features of the C language, and the C Library. In particular, we require pointer arguments to variable argument functions to be `private`. This caused no problems when SharC was applied to the benchmarks in Section 5. Further, we stipulate that C Library calls require pointer arguments be `private`. However, SharC also supports trusted annotations that summarize the read/write behavior of library calls. When the read/write behavior of a library call is summarized for an argument, the call may accept an actual argument in any sharing mode except for `locked`. In particular, for a `dynamic` actual, the read/write summary tells how to update the reader/writer sets for the object, and a `readonly` actual can be safely passed when there is a read summary.

4.5 Limitations

SharC has a couple of limitations. First, false race reports may result from false sharing, and from the use of custom memory allocators. Since we track races at a 16-byte granularity, races may be reported for two separate objects that are close together, but used in a non-racy way. To alleviate this problem, SharC ensures that malloc allocates objects on a 16-byte boundary. If a program's custom memory allocator transfers unused memory between threads, or does not allocate on 16-byte boundaries, SharC may incorrectly report races. In the future, we will provide support for making SharC understand custom allocators.

Second, our analysis is dynamic, so it will only detect sharing strategy violations over a limited number of paths, and only for thread schedules that actually occur on a concrete run of the program. The advantage of the dynamic analysis is, of course, that errant program behavior will be detected when it occurs, rather than at some later time.

5. Evaluation

We applied SharC to 6 multithreaded C programs totalling over 600k lines of code. Two of the programs were over 100k lines. These programs use threads in a variety of ways: some use threads for performance, whereas others use threads to hide I/O latency. Further, one benchmark is a server that spawns a thread for each client.

We found the following procedure for applying SharC to be expedient. Minimal changes are made to the source until the inference stage no longer results in ill-formed types. This involves removing casts that that incorrectly strip off our type qualifiers. Then, we run the program and inspect the error reports. These are usually sufficient to tell how data is shared in the program, and we use them to decide what objects are protected by which locks, and to note where the sharing mode of objects changes (typically, SharC's sharing cast suggestions can be applied as is).

The goal of these experiments is to demonstrate that our approach is practical. In particular, it requires few enough code changes, and incurs low enough overhead that it could be used in production systems. We found no serious errors[2] in out benchmarks because our tests only sought to exercise typical runs of the programs — we did not perform thorough regression testing.

Table 1 summarizes our experience using SharC. The reported runtimes are averages of 50 runs of each of the benchmarks. Memory overhead was measured by recording the number of minor pagefaults[3] incurred by each benchmark. All tests were run on a machine with a dual core 2GHz Intel Xeon processor with 2GB of memory. A total of 60 annotations, and 122 other code changes were required for the 600k lines of code. On average, SharC incurred a performance overhead of 9.2%, and a memory overhead of 26.1%.

The pfscan benchmark is a tool that spawns multiple threads for searching through files. It combines some features of grep and find. One thread finds all the paths that must be searched, and an arbitrary number of threads take paths off of a shared queue protected with a mutex and search files at those paths. Our test searched for a string in all files in an author's home directory. We found that running the test several times allowed all files to be held in the operating system's buffer cache, and so we were able to eliminate file systems effects in measuring the overhead.

The aget benchmark is a download accelerator. It spawns several threads that each download pieces of a file. We measured performance by downloading a Linux kernel tarball. The program was network bound, and so the overhead created by SharC was not measurable.

The pbzip2 benchmark is a parallel implementation of the block-based bzip2 compression algorithm. The benchmark consisted of using three worker threads to compress a 4MB file. The pbzip2 benchmark has threads for file I/O, and an arbitrary number of threads for (de)compressing data blocks, which the file-reader thread arranges into a shared queue. The functions that perform the (de)compression assume that they have ownership of the blocks, and so we annotate their arguments as `private`. One benign race was found in a flag that is used to signal that reading from the input file has finished. At worst, a thread might yield an extra time before exiting.

The dillo benchmark is a web browser that aims to be small and fast. We measured the overhead of SharC by starting the browser, requesting a sequence of 8 URLs, and then closing the browser. The dillo benchmark uses threads to hide the latency of DNS lookup. It keeps a shared queue of the outstanding requests. Four worker threads read requests from the queue and initiate calls to `gethostbyname`. Several functions called from the worker threads assume that they own request data, so the arguments to these functions were annotated `private`. The memory overhead for dillo is higher because integers are cast to pointer type, and SharC infers they need to be reference counted. These bogus pointers are never dereferenced, but we incur minor pagefaults when their reference counts are adjusted. We suspect that this issue could be addressed if the programmer annotates the pointers that only store bogus values.

The fftw benchmark performs 32 random FFT's as generated by the benchmarking tool distributed with The Fastest Fourier Transform in the West [11]. The fftw benchmark computes by dividing arrays among a fixed number of worker threads. Ownership of parts of the array are transferred to each thread, and then reclaimed when the threads are finished. The functions that compute over the partial arrays assume that they own that memory, so it was only necessary to annotate those arguments as `private`.

The stunnel benchmark is a tool that allows the encryption of arbitrary TCP connections. It creates a thread for each client that it serves. The main thread initializes data for each client thread before spawning them. There are also global flags and counters, which are protected by locks. Stunnel uses the OpenSSL library, so it is also necessary to process it with SharC. Even though OpenSSL is not concurrent itself, SharC is able to verify that there are no thread-safety issues with its use by stunnel in our tests. Our experiments

[2] By "serious error" we mean a sharing strategy violation that causes unintended results.

[3] The number of minor pagefaults indicates the number of times the operating system kernel maps a page of the process's virtual address space to a physical frame of memory. It is a rough measure of memory usage.

Benchmark					Time		Pagefaults		% `dynamic`
Name	Threads	Lines	Annots.	Changes	Orig.	SharC	Orig.	SharC	Accesses
pfscan	3	1.1k	8	11	1.84s	12%	21k	0.8%	80.0%
aget	3	1.1k	7	7	n/a	n/a	0.4k	30.8%	8.7%
pbzip2	5	10k	10	36	0.83s	11%	10k	1.6%	~0.0 %
dillo	4	49k	8	8	0.69s	14%	2.6k	78.8%	31.7 %
fftw	3	197k	7	39	44.1s	7%	63k	1.2%	0.2 %
stunnel	3	361k	20	22	0.39s	2%	0.5k	43.5%	~0.0%

Table 1. Benchmarks for SharC. For each test we show the maximum number of threads running concurrently(Threads), the size of the benchmark including comments (Lines), the number of annotations we added (Annots.) and other changes required (Changes). We also report the time and memory overhead caused by SharC along with the proportion of memory accesses to `dynamic` objects.

with stunnel involved encrypting three simultaneous connections to a simple echo server with each client sending and receiving 500 messages.

5.1 Conversion Effort

Our experiments on the programs above did not require extremely deep understanding. Locks tended to be declared near the objects they protected, threading-related code tended to be in a file called "thread.c", `private` annotations were made close–both textually and in the call graph–to thread creation calls, and so forth. As in Section 2.1's example, SharC's error reports were often helpful in guiding annotation insertion. Also, SharC infers a reasonable default for omitted annotations. Therefore, we had no insurmountable problems in adding the few needed annotations. Time required to read and annotate relevant code varied between 2 and 4 hours. Time for reading and annotating larger codes did not grow proportionally because threading-related code tended to be concentrated in one place. Since the annotation burden is low, we do not believe that automating the annotation process would have a substantial benefit.

6. Related Work

The work most closely related to our own are tools that attempt to find data races in concurrent programs. Whereas SharC attempts to identify violations of a sharing strategy, race detectors simply look for unsynchronized access to memory. There are several different approaches to race detection. We classify the approaches as static, dynamic, and model-checking.

6.1 Static Race Detection and Model Checking

There has been much work on statically analyzing data races and atomicity in Java programs [2, 21, 17]. We have used some ideas about ownership types from these works in our own type system, and believe that atomicity is an important concern requiring further attention in dynamic analysis of large real-world legacy C programs.

Cyclone [13] allows the programmer to write type annotations that enable its compiler to statically guarantee race-freedom. The difficulty in applying Cyclone's analysis to existing multithreaded programs lies in translation to a program with Cyclone's annotations and concurrency primitives. Our system requires annotations, but they are far less pervasive.

Relay [25], and RacerX [7] are static lockset based analyses for C code that scale to large real world programs including the Linux kernel. However, both tools are unsound, and require significant post-processing of warnings to achieve useful results. Locksmith [19], on the other hand, is a sound static race detector for C. It statically infers the correlation of memory locations to the locks that protect them. If a shared location is not consistently protected by the same lock, a race is reported. Locksmith also does a sharing analysis similar to our own as an optimization. Unfortunately, Locksmith runs out of resources while analyzing larger programs.

Some of these techniques have scaled up to many hundreds of thousands of lines of code and have uncovered serious problems in real world software. Further, some of these techniques, especially the ones for Java, also achieve manageable false positive rates. For development cultures in which programmers are encouraged to use the results of static analysis, these techniques are probably an appropriate choice. However, testing is already widely used, and so low overhead dynamic analysis will be the least cost path to race detection for many people.

Sen and Agha [23] perform explicit path model checking of multithreaded programs by forcing races detected on a concrete run to permute by altering the thread schedule, and by solving symbolic constraints to generate inputs that force the program down new paths. KISS and the work of Henzinger et. al. can also find errors in concurrent programs with model checking [20, 14].

6.2 Dynamic Race Detection

Eraser [22] popularized the dynamic lockset algorithm for race detection. The goal of the lockset algorithm is to ensure that every shared location is protected by a lock. Eraser monitors every memory read and write in the program through binary instrumentation, and tracks the state of each location in memory. The states that a location can inhabit model common idioms such as initialization before sharing, read-sharing, read-write locking, and so forth. Eraser is able to analyze large real-world programs, but it incurs a 10x-30x runtime overhead. Further, the state diagram used to determine when a race might be occurring may not be an accurate model of the data sharing protocol in a program. This inaccuracy leads to false positives.

Improvements to the lockset algorithm use Lamport's happens-before relation to track thread-ownership changes. This reduces false positives due the lockset state diagram failing to model signaling between threads, among other things. Additionally, some dynamic race detectors perform preliminary static analysis to improve performance. Analyses using these improvements have achieved lower false positive rates and better performance [1, 8, 3]. For Java, the overhead has been reduced to 13%-42% [4]. Goldilocks integrates race detection into the Java runtime [6]. Racetrack integrates race detection into the CLR [26], and achieves overhead in the range 1.07x-3x for .Net languages, with the low end corresponding to non-memory-intensive programs. Using the happens-before relation and more complicated state diagrams to model additional data sharing schemes reduces false positives, but our system is the first to attack the root of the problem by modeling ownership transfer directly.

7. Conclusion

We have presented SharC, the first tool that allows programmers to specify and verify (through static and dynamic checks) the data sharing strategy used in multithreaded C programs. We have shown the promise and practicality of SharC by applying it to over 600k lines of legacy C code with few annotations and performance overhead under 10% on average.

SharC can still be improved. In particular, its runtime race detection should be able to handle a larger number of threads with low overhead. SharC may also need new sharing modes to better support existing sharing strategies (e.g., more support for locks), and to model new sharing strategies (e.g., transactional memory).

7.1 Future Work

It was mentioned in Section 4 that for full soundness, SharC must rely on some external tool, which would provide type- and memory-safety. For that reason, we are currently integrating SharC with the Deputy [5], and Heapsafe [12] tools. Deputy provides for type- and memory-safety, and Heapsafe provides for deallocation safety(i.e. the absense of dangling references.) It is interesting to note that Deputy and Heapsafe by themselves are unsound in the presence of sharing strategy violations, but when combined with SharC they provide an incremental pathway to type- and memory-safe concurrent C programs.

References

[1] Agarwal, R., Sasturkar, A., Wang, L., and Stoller, S. D. Optimized run-time race detection and atomicity checking using partial discovered types. In *ASE'05*.

[2] Boyapati, C., Lee, R., and Rinard, M. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA'02*, pp. 211–230.

[3] Cheng, G.-I., Feng, M., Leiserson, C. E., Randall, K. H., and Stark, A. F. Detecting data races in Cilk programs that use locks. In *SPAA'98*, pp. 298–309.

[4] Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI'02*, pp. 258–269.

[5] Condit, J., Harren, M., Anderson, Z., Gay, D., and Necula, G. Dependent types for low-level programming. In *ESOP'07*.

[6] Elmas, T., Qadeer, S., and Tasiran, S. Goldilocks: a race and transaction-aware Java runtime. In *PLDI'07*, pp. 245–255.

[7] Engler, D., and Ashcraft, K. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP'03*, pp. 237–252.

[8] Flanagan, C., and Freund, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL'04*, pp. 256–267.

[9] Foster, J. S., Fahndrich, M., and Aiken, A. A theory of type qualifiers. In *PLDI'99*, pp. 192–203.

[10] freedesktop.org. Gstreamer: Open source multimedia framework. http://gstreamer.freedesktop.org/.

[11] Frigo, M. A fast Fourier transform compiler. In *PLDI'99*, pp. 169–180.

[12] Gay, D., Ennals, R., and Brewer, E. Safe manual memory management. In *ISMM'07* (New York, NY, USA, 2007), ACM, pp. 2–14.

[13] Grossman, D. Type-safe multithreading in Cyclone.

[14] Henzinger, T. A., Jhala, R., and Majumdar, R. Race checking by context inference. In *PLDI'04*, pp. 1–13.

[15] Intel Corp. *Intel 64 Architecture Memory Ordering White Paper*, 1.0 ed., August 2007.

[16] Levanoni, Y., and Petrank, E. An on-the-fly reference-counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems 28*, 1 (2006), 1–69.

[17] Naik, M., Aiken, A., and Whaley, J. Effective static race detection for Java. In *PLDI'06*, pp. 308–319.

[18] Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems 27*, 3 (May 2005).

[19] Pratikakis, P., Foster, J. S., and Hicks, M. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI'06*, pp. 320–331.

[20] Qadeer, S., and Wu, D. KISS: keep it simple and sequential. In *PLDI'04*, pp. 14–24.

[21] Sasturkar, A., Agarwal, R., Wang, L., and Stoller, S. D. Automated type-based analysis of data races and atomicity. In *PPoPP'05*, pp. 83–94.

[22] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP'97*, pp. 27–37.

[23] Sen, K., and Agha, G. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference* (2006), pp. 166–182.

[24] US-CERT. Technical cyber security alerts. http://www.us-cert.gov/cas/techalerts/index.html.

[25] Voung, J. W., Jhala, R., and Lerner, S. RELAY: static race detection on millions of lines of code. In *ESEC-FSE'07*, pp. 205–214.

[26] Yu, Y., Rodeheffer, T., and Chen, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP'05*, pp. 221–234.

A. Soundness Proof

Our basic approach is to prove, by induction over the operational semantic steps that the following three properties hold at all times:

- The statements in all threads are well-typed.
- The memory and thread environments are consistent (Definition 1).
- The statements in all threads are well-checked (necessary runtime checks have or will be performed).

From these properties it is easy to prove that private cells are only accessed by their owner and that dynamic cells are only accessed by threads in the cell's reader (for reads) or writer (for writes) sets.

Before proceeding with the proof, we start with a few preliminaries and give formal definitions of well-typedness (Section A.2 – Figure 4 is not applicable as it is a rewriting rather than type-checking system), well-checkedness and the conditions under which a runtime check is valid (Section A.3 – again, Figure 5 is a system for performing runtime checks, not for verifying that they hold). Finally, to make the proof clearer we split our earlier definition of consistency into two (Section A.4): environmental consistency checks the link between an environment and a memory, while memory consistency checks the link between memory cells of reference type.

The proof itself consists of a few general-usage lemmas (Section A.5), proofs that single-thread steps, thread creation and thread destruction preserve the three properties (Sections A.6 through A.9), and a final section that puts all the pieces together to prove that the statically declared sharing modes are respected (Section A.10).

A.1 Preliminaries

For simplicity, we assume all variables have distinct names, and that the type of variable x is τ_x. We use *address* to refer to an element of the domain of a memory, and note that 0 is never a valid

address ($0 \notin \text{dom}(M)$). We use *cell* to refer to an actual memory element $M(a)$. An *l-value* is an expression that denotes the address of a particular cell.

We define three properties of cells, unaddressed, privaddr and stable, whose use helps prove that sharing casts in one thread do not invalidate assumptions made about l-values in other threads (see Sections A.2 and A.3). In particular, as we will prove below (Lemma 6) a stable address is unaffected by sharing casts in other threads.

Definition 2. Cell $M(a)$ is *not addressable*, written unaddressed(M, a) if $\nexists b \in \text{dom}(M).M_v(b) = a \wedge M_\tau(b) = m \text{ ref } \tau$.

Definition 3. Cell $M(a)$ is *privately addressed* by thread i, written privaddr(M, i, a) if $\exists b \in \text{dom}(M).M_v(b) = a \wedge M_\tau(b) = \texttt{private ref } \tau \wedge M_o(b) = i$.

Definition 4. Cell $M(a)$ is a *stable cell* of thread i, written stable(M, i, a) if unaddressed$(M, a) \vee$ privaddr(M, i, a).

A.2 Runtime Typing

To show that our operational semantics preserve types, we show that programs remain well-typed at all points during execution. To do this, we need typing rules that do not rewrite expressions and that give types to addresses (a) and runtime-only statements (**skip**, **done**, **fail**). These typing rules are given in Figure 10 and closely match the typing judgments of Figure 4. Note that the second statement in a sequence (SEQ-R) is checked in an empty memory, so cannot contain any addresses (a).

The stable(M, id, a) precondition in ADDRESS-R is necessary to prove that executions of sharing casts in other threads do not change types in partially evaluated assignments.

A.3 Tracking Runtime Check Validity

The soundness proof needs to track whether a thread is *well-checked*, i.e., necessary runtime checks either hold or will be performed. In particular, it must deal with the fact that between the time when thread i executes a runtime check ω and the time thread i performs the assignment that relies on ω, the other threads may have performed arbitrary computation, so the continuing validity of ω is not immediately obvious. Thus we need to be able to assert the continuing validity of a runtime check without modifying memory. For this purpose, Figure 11 defines the $M, i \models \omega(a)$ (compare with Figure 5 which performs a runtime check, modifying M). The stable(M, id, a) precondition is necessary to prove that executions of sharing casts in other threads do not invalidate runtime checks. We can now define the checked property that ensures that a thread is well-checked.

Definition 5. A statement s of thread id is *well-checked* in memory M and environment E, written checked(M, E, id, s), if the runtime checks necessary for s have not yet been executed or already hold. Formally, let $s' = l := e$ when $\omega_1(\ell_1), \omega_2(\ell_2) \ldots, \omega_n(\ell_n)$ $(n \geq 0)$ be an assignment in s. We define Ω, the set of runtime checks available in thread id at s':. If $s = s'; s''$ any valid check can be used:

$$\Omega = \{\omega_1(\ell_1), \omega_2(\ell_2) \ldots, \omega_n(\ell_n)\} \cup \{\omega(a) | M, id \models \omega(a)\}$$

otherwise the assignment must contain all necessary runtime checks:

$$\Omega = \{\omega_1(\ell_1), \omega_2(\ell_2) \ldots, \omega_n(\ell_n)\}$$

The ensured property asserts that a particular runtime check holds or will be performed:

$$\text{ensured}(\epsilon)$$
$$\omega(\ell) \in \Omega \Rightarrow \text{ensured}(\omega(\ell))$$
$$\omega(a) \in \Omega \wedge M, E : \ell \xrightarrow{\ell} a \Rightarrow \text{ensured}(\omega(\ell))$$

$M, id \models \ell : \tau$ — In memory M and thread id, ℓ is a well-typed lvalue with type τ.

$$\frac{}{M, id \models x : \tau_x}\ (\text{NAME-R}) \qquad \frac{\tau_x = \texttt{private ref } \tau}{M, id \models *x : \tau}\ (\text{DEREF-R})$$

$$\frac{M_\tau(a) = \texttt{private } \sigma \Rightarrow M_o(a) = id \qquad \text{stable}(M, id, a)}{M, id \models a : M_\tau(a)}\ (\text{ADDRESS-R})$$

$M, id \models s$ — In memory M and thread id, statement s is well-typed.

$$\frac{M, id \models s_1 \qquad [], 0 \models s_2}{M, id \models s_1; s_2}\ (\text{SEQ-R}) \qquad \frac{}{M, id \models \text{spawn } f()}\ (\text{SPAWN-R})$$

$$\frac{}{M, id \models \textbf{done}}\ (\text{DONE-R}) \qquad \frac{}{M, id \models \textbf{skip}}\ (\text{SKIP-R}) \qquad \frac{}{M, id \models \textbf{fail}}\ (\text{FAIL-R})$$

$$\frac{M, id \models \ell_i : \tau_i \qquad M, id \models \ell := e}{M, id \models \ell := e \text{ when } \omega_1(\ell_1), \ldots, \omega_n(\ell_n)}\ (\text{RUNTIME-CHECK-R})$$

$$\frac{M, id \models \ell : m \text{ int}}{M, id \models \ell := n}\ (\text{CONSTANT-ASSIGN-R}) \qquad \frac{M, id \models \ell : m \text{ ref } \tau}{M, id \models \ell := \text{null}}\ (\text{NULL-ASSIGN-R})$$

$$\frac{M, id \models \ell : m \text{ ref } \tau}{M, id \models \ell := \text{new}_\tau}\ (\text{NEW-ASSIGN-R})$$

$$\frac{M, id \models \ell_1 : m_1\ \sigma \qquad M, id \models \ell_2 : m_2\ \sigma}{M, id \models \ell_1 := \ell_2}\ (\text{ASSIGN-R})$$

$$\frac{\begin{array}{c} M, id \models \ell_1 : m \text{ ref } (m_1\ \sigma) \\ M, id \models \ell_2 : \texttt{private ref } (m_2\ \sigma) \qquad \tau = m_1\ \sigma \end{array}}{M, id \models \ell_1 := \text{scast}_\tau\ \ell_2}\ (\text{CAST-ASSIGN-R})$$

Figure 10. Runtime typing judgments.

$$\frac{\text{stable}(M, id, a) \qquad M_W(a) - \{id\} = \emptyset \qquad id \in M_R(a)}{M, id \models \mathit{chkread}(a)}$$

$$\frac{\text{stable}(M, id, a) \qquad M_R(a) - \{id\} = \emptyset \qquad M_W(a) = \{id\}}{M, id \models \mathit{chkwrite}(a)}$$

$$\frac{\text{stable}(M, id, a) \qquad |\{b | M_v(b) = a \wedge M_\tau(b) = m \text{ ref } \tau\}| = 1}{M, id \models \mathit{oneref}(a)}$$

Figure 11. Semantics for confirming runtime checks.

For checked to hold, we require that

- if $M, id \models \ell : m\ \sigma$ then ensured(W(ℓ, m))
- if $e = \ell'$ and $M, id \models \ell' : m'\ \sigma'$ then ensured(R(ℓ', m'))
- if $e = \text{scast}_\tau\ x$ then ensured($oneref(*x)$)
- if $e = \text{scast}_\tau\ a$ then ensured($oneref(M_v(a))$)

A.4 Consistency

To ensure that programs remain type-safe, we need to know that types and owners in the memory are consistent, and that each thread's environment (which maps variable names to the address with the variable's value) is consistent with the memory. Furthermore, to avoid problems with sharing casts changing the types of variables, we must assert that variables are unadressable.

For clarity, we separate these two requirements into two properties, *environmental consistency* and *memory consistency*. All our lemmas and theorems require memory and/or environmental consistency as a precondition.

Definition 6. *Environmental consistency.* M is *consistent with the environment* E *of thread* id, written $E, id \models M$, if types and owners are consistent between E, M and the static semantics, and variable addresses are never stored in M. Formally, for all variables x:

1. $M_\tau(E(x)) = \tau_x$ (variable types are preserved)
2. $x \notin \text{dom}(G) \Rightarrow M_o(E(x)) = id$ (local variables are owned by their thread)
3. unaddressed$(M, E(x))$ (variables are not addressable)

Definition 7. *Memory consistency.* M is *consistent*, written $\models M$, if types and owners are consistent when one cell refers to another within M. Furthermore, the readers and writers sets in M must have legal values, and the current runtime checks must hold. Formally, for all addresses a:

1. $|M_W(a)| \leq 1$ (no more than one writer)
2. $M_W(a) \neq \emptyset \Rightarrow M_R(a) \subseteq M_W(a)$
   (no other readers than the writer)

and for all addresses b with non-zero value $c = M_v(b)$:

3 $M_\tau(b) = m$ ref $\tau \Rightarrow M_\tau(c) = \tau$ (types are consistent)
4 $M_\tau(b) =$ `private` ref (`private` σ) $\Rightarrow M_o(b) = M_o(c)$ (owners are consistent)

In the rest of this proof we sometimes use *type consistency* to refer to clause 3 of memory consitency and *owner consistency* to refer to clause 4.

A.5 Basic Properties

We prove five useful basic properties:

- Single-thread steps preserve unaddressed cells.
- Other-thread steps preserve stable addresses.
- Private cells are only modified by their owner.
- Other-thread steps preserve l-value types.
- Sharing casts do not affect stable cells.

Lemma 1. *Single-thread steps preserve unaddressed cells.* If

$$\models M \qquad M, id \models s \qquad M, E, id : s \rightarrow M', f, s'$$

then unaddressed$(M, b) \Rightarrow$ unaddressed(M', b)
Proof: by case inspection on s.

- Trivial for non-assignment steps, as

$$\forall a \in \text{dom}(M').M_\tau(a) = M'_\tau(a) \wedge M_v(b) = M'_v(b)$$

- $a :=$ null: $b \neq 0$ ($b \in \text{dom}(M)$ and $0 \notin \text{dom}(M)$) so $M'_v(a)$ cannot address b.
- $a := n$: from $M, id \models s$, $M_\tau(a) = m$ int, so $M'_v(a)$ cannot address b.
- $a :=$ new$_\tau$: max(dom(M)) + 1 $\notin$ dom(M), while $b \in$ dom(M), so $M'_v(a)$ cannot address b.
- $a_1 := a_2$: from $M, id \models s$, $M'_\tau(a_1)$ is a reference iff $M_\tau(a_2)$ is a reference. Therefore, $M'_v(a_1) = M_v(a_2)$ cannot address b.
- $a_1 := \text{scast}_{m_1\ \sigma}\ a_2$: $M'_v(a_2)$ cannot address b (see $a :=$ null), and neither can $M'_v(a_1)$ (see $a_1 := a_2$). Finally, from $M, id \models a_2$: `private` ref $(m_2\ \sigma)$ and the type consistency of M we know that $M_\tau(v_2) = m_2\ \sigma$ is a reference iff $M'_\tau(v_2) = m_1\ \sigma$ is a reference, so $M'_v(v_2)$ cannot address b.

Lemma 2. If $i \neq j$, $M, i \models a : \tau$ and for all $c \in \text{dom}(M')$

$$c \neq a \Rightarrow M'_\tau(c) = M_\tau(c) \wedge M'_o(c) = M_o(c) \wedge M'_v(c) = M_v(c)$$

then privaddr$(M, j, b) \Rightarrow$ privaddr(M', j, b).
Proof: Let d be a witness of privaddr(M, j, b), i.e.,

$$M_v(d) = b \wedge M_\tau(d) = \texttt{private ref}\ \tau' \wedge M_o(d) = j$$

Assume $a = d$. Then $M_\tau(a) =$ `private` ref τ' and by ADDRESS-R, $M_o(a) = i$, a contradiction, so $a \neq d$. Therefore d is a a witness for privaddr(M', j, b).

Lemma 3. *Other-thread steps preserve stable addresses.* If

$$i \neq j \qquad \models M \qquad M, i \models s \qquad M, E, i : s \rightarrow M', f, s'$$

then stable$(M, j, b) \Rightarrow$ stable(M', j, b).
Proof: Lemma 1 shows that unaddressed$(M, b) \Rightarrow$ unaddressed(M', b) so it suffices to prove privaddr$(M, j, b) \Rightarrow$ privaddr(M', j, b). We proceed by case inspection on s.

- Trivial for non-assignment steps by Lemma 2.
- $a :=$ null, $a := n$, $a := a_2$, $a :=$ new$_\tau$: as $M, i \models a : \tau$, Lemma 2 shows that privaddr(M', j, b).
- $a_1 :=$ scast$_\tau\ a_2$: as $M, i \models a_1 : m$ ref$(m_1\ \sigma)$ and $M, i \models a_2$: `private` ref $(m_2\ \sigma)$, two applications of Lemma 2 show that privaddr(M'', j, b) where $M'' = M[a_1 \xrightarrow{v} v_2, a_2 \xrightarrow{v} 0]$.
  To show privaddr(M', j, b), let d be a witness of privaddr(M'', j, b):

  $$M''_v(d) = b \wedge M''_\tau(d) = \texttt{private ref}\ \tau' \wedge M''_o(d) = j$$

  We show by contradiction that $d \neq v_2$, and hence that d is a witness for privaddr(M', j, b). Assume $d = v_2$. Then $M''_\tau(v_2) =$ `private` ref τ'. From ADDRESS-R, $M_o(a_2) = i$. By the type consistency of M, $m_2\ \sigma = M''_\tau(v_2)$, so $m_2 =$ `private` and hence by the owner consistency of M, $j = M''_o(v_2) = M''_o(a_2) = i$, a contradiction.

Lemma 4. *Private cells are only modified by their owner.* If

$$M, i \models s \qquad M, E, i : s \rightarrow M', f, s'$$
$$i \neq j \qquad M_\tau(a) = \texttt{private}\ \sigma \qquad M_o(a) = j$$

then $M_v(a) = M'_v(a)$
Proof: By case inspection on s. Only assignments modify M_v. Consider an assignment s'' that modifies a. By inspection, we see that $M, i \models s''$ implies $M, i \models a : \tau$. From ADDRESS-R's preconditions, we conclude that $M_o(a) = i$, a contradiction. Therefore, no assignments modify a and $M_v(a) = M'_v(a)$.

Lemma 5. *Other-thread steps preserve l-value types.* If

$$i \neq j \qquad \models M \qquad M, i \models s_i \qquad \text{checked}(M, E_i, i, s_i)$$
$$M, E_i, i : s_i \rightarrow M', f, s'_i$$

then $M, j \models \ell : \tau \Rightarrow M', j \models \ell : \tau'$

Proof: The only non-trivial case is $\ell = a$. First, we note that stable(M', j, a) follows from Lemma 3. Second, the only step that can change types or owners is $a_1 := \text{scast}_\tau\ a_2$. By Lemma 6, $a \neq M_v(a_2)$ so the type and owner of a are preserved. Thus $M', j \models a : \tau$.

Lemma 6. *Sharing casts do not affect stable cells.* If

$$\text{checked}(M, E, i, s) \quad \models M \quad M, i \models a_1 := \text{scast}_\tau\ a_2$$
$$i \neq j \quad \text{stable}(M, j, a)$$

then $M_v(a_2) \neq a$.
Proof: Assuming $a = M_v(a_2)$ implies a contradiction. First, we note that unaddressed(M, a) does not hold. Therefore, privaddr(M, j, a) must hold with some witness b such that

$$M_v(b) = a \wedge M_\tau(b) = \texttt{private ref}\ \tau' \wedge M_o(b) = j$$

From checked(M, E, i, s), we know that $M, i \models \mathit{oneref}(a)$, so a_2 is the only reference to a, i.e. $a_2 = b$. However

$$M, i \models a_2 : \texttt{private ref}\ \tau'$$

so $j = M_o(b) = M_o(a_2) = i$, a contradiction.

A.6 Preserving Runtime Checks

We show that threads start with all necessary runtime checks, and that no single-thread step can cause a runtime check to get lost.

Lemma 7. *Necessary runtime checks are inserted.* If

$$\Gamma \vdash s \Rightarrow s'$$

then checked(M, E, id, s').
Proof: By inspection of the rules in Figure 4.

Lemma 8. *Same-thread steps preserve runtime checks.* If

$$M, id \models s \qquad M, E, id : s \to M', f, s'$$

then checked$(M, E, id, s) \Rightarrow$ checked(M', E, id, s').
Proof: By case inspection on s. We summarize the interesting cases:

- An l-value evaluation $(M, E : \ell \xrightarrow{\ell} a)$ may replace $\omega(\ell)$ by $\omega(a)$, leaving $M' = M$. This preserves checked because of the third clause of ensured.
- An l-value evaluation $(M, E : \ell \xrightarrow{\ell} a)$ may transform the first statement of s from $\ell := e$ into $a := e$, leaving $M' = M$. We consider the non-trivial case where $M_\tau(a) = \texttt{dynamic}\ \sigma$. By assumption, ensured$(\mathit{chkwrite}(\ell))$ and so, as the assignment no longer has any when clauses, $M, id \models \mathit{chkwrite}(a)$. Therefore $M', id \models \mathit{chkwrite}(a)$ and checked is preserved.
  The same argument applies to the transformation of $a' := \ell$ into $a' := a$.
- Runtime-check execution removes $\omega_1(a)$ from the first statement of s. We know that $M, id \models \omega_1(a) \Rightarrow M'$ and (from $M, id \models a : \tau$) stable(M, id, a). Hence $M', id \models \omega(a)$. Furthermore, $M, id \models \omega(c)$ implies $M', id \models \omega(c)$ as *oneref* checks are clearly unaffected by the changes between M and M', and the updates to M_R and M_W cannot invalidate existing *chkread*, *chkwrite* checks. Thus, the set Ω used in checked is unchanged, so checked is preserved.
- After execution, an assignemnt is replaced by **skip**, so places no requirements on checked(M', E, id, s'). The other assignments in s' were not the first assignments of s so contain no addresses and were checked without reference to M. They are therefore clearly still checked.

Lemma 9. *Other-thread steps preserve runtime checks.* If

$$\models M \quad M, i \models s_i \quad \text{checked}(M, E_i, i, s_i) \quad M, E_i, i : s_i \to M', f, s_i'$$
$$i \neq j \quad E_j, j \models M \quad M, j \models s_j$$

then checked$(M, E_j, j, s_j) \Rightarrow$ checked(M', E_j, j, s_j).
Proof: We prove this lemma by showing the following four properties whose combination ensures that shows checked$(M, E_j, j, s_j) \Rightarrow$ checked(M', E_j, j, s_j):

$$M, E_j : \ell \xrightarrow{\ell} a \Rightarrow M', E_j : \ell \xrightarrow{\ell} a$$
$$M, j \models \ell : \tau \Rightarrow M', j \models \ell : \tau$$
$$s_j = a_1 := \text{scast}_\tau\ a_2; s \Rightarrow M_v(a_2) = M'_v(a_2)$$
$$M, j \models \omega(c) \Rightarrow M', j \models \omega(c)$$

$M, E_j : \ell \xrightarrow{\ell} a \Rightarrow M', E_j : \ell \xrightarrow{\ell} a$ is trivial for $\ell = x$. When $\ell = *x$, $E_j, j \models M$ implies $M_o(E_j(x)) = j$, so by Lemma 4 $M'_v(E_j(x)) = M_v(E_j(x))$.

$M, j \models \ell : \tau \Rightarrow M', j \models \ell : \tau$ follows directly from Lemma 5.

If $s_j = a_1 := \text{scast}_\tau\ a_2; s$ then from $M, j \models s_j$, $M_o(a_2) = j$, so Lemma 4 implies $M_v(a_2) = M'_v(a_2)$.

Assume $M, j \models \omega(c)$. $M', j \models \omega(c)$ is trivial when $M = M'$. Lemma 3 guarantees stable(M', j, c). To prove the remaining requirements of $M', j \models \omega(c)$, we analyse the cases where a statement executed by thread i leaves $M \neq M'$:

- $\ell := e$ when $\omega_1(a), \omega_2(\ell_2) \ldots, \omega_n(\ell_n)$: as we noted in Lemma 8, successful runtime checks do not invalidate other checks, so $M', j \models \omega(c)$.
- In all assignment statements except scast, $M'_R(c) = M_R(c)$ and $M'_W(c) = M_W(c)$, so *chkread* and *chkwrite* checks remain valid.
- For $\omega = \mathit{oneref}$, we know there is exactly one b such that $b \in \text{dom}(M).M_v(b) = c \wedge M_\tau(b) = m$ ref τ. Furthermore, we know that $m = \texttt{private} \wedge M_o(b) = j$. We show that if a is an assignment target then $a \neq b$ and that $M'_v(a)$ is not a reference to c, and hence that *oneref* remains valid.
  Assume $a = b$. From $M, id \models s$ and type consistency, we get

  $$M, id \models a : \texttt{private ref}\ \tau$$

  and hence $M_o(a) = id$ which contradicts $M_o(b) = j$.
  For $a :=$ null: $M'_v(a) = 0 \neq c$.
  For $a := n$: $M'_\tau(a) = m$ int so $M'_v(a)$ is not a reference to c.
  For $a := \text{new}_\tau$: $M'_v(a) = \max(\text{dom}(M)) + 1 \neq c$.
  For $a := a_2$: following the same argument as for a, we obtain $a_2 \neq b$. If $M'_\tau(a) = m$ int then $M'_v(a)$ is not a reference to c, otherwise $M_\tau(a_2) = m$ ref σ and (from the uniqueness of b) $M'_v(a) = M_v(a_2) \neq c$.
- $a_1 := \text{scast}_\tau\ a_2$: let $v_2 = M_v(a_2)$. Lemma 6 implies that $c \neq v_2$. *chkread* and *chkwrite* checks on addresses for $c \neq v_2$ are preserved as $M_R(c) = M'_R(c)$, $M'_W(c) = M_W(c)$. *oneref* checks on these addresses are also preserved following the same arguments as for the $a :=$ null and $a_1 := a_2$ statements.

A.7 Preserving Type Safety

We prove that single steps of the operational semantics preseve runtime type safety in two separate lemmas, one for same-thread steps and one for other-thread steps.

Lemma 10. *Same-thread steps preserve runtime type safety.* If

$$\models M \qquad E, id \models M \qquad M, E, id : s \to M', f, s'$$

then $M, id \models s \Rightarrow M', id \models s'$.
Proof: By case inspection of s.

- Steps due to l-value evaluation $(M, E : \ell \xrightarrow{\ell} \ell')$ leave $M' = M$, and replace x or $*x$ by an address (a). We show that the ADDRESS-R rule is satisfied by a and that a has the same type as the original l-value, and hence $M, id \models s \Rightarrow M', id \models s'$.
  Consider first $\ell = x$, $a = E(x)$: by rule NAME-R, the type of ℓ is τ_x. From $E, id \models M$ we obtain $M_\tau(E(x)) = \tau_x$, $M_o(E(x)) = id$

and unaddressed$(M, E(x))$ so ADDRESS-R holds and the type is preserved.
When $\ell = *x$ and $a = M_v(E(x))$: by rule DEREF-R $\tau_x = \texttt{private ref}\ \tau$ and the type of ℓ is τ. We know that $a \neq 0$ (if $a = 0$ then the thread switches to the **fail** statement and the lemma trivially holds). From $E, id \models M$ and the type consistency of M, we obtain that $M_\tau(E(x)) = \texttt{private ref}\ \tau$, $M_\tau(a) = \tau$ and $M_o(E(x)) = id$. If $\tau = \texttt{private}\ \sigma$ then the owner consistency of M implies $M_o(a) = id$. Finally, $E(x)$ is a witness for privaddr(M, id, a) so ADDRESS-R holds and the type is preserved.

- Many operational semantics rules replace $s_1; s_2$ by **skip**; s_2 or **fail**; s_2. As $[], 0 \models s_2$ (SEQ-R) $M', id \models$ **skip** or **fail**; s_2 for any M'.
  Essentially the same argument applies to the $M, id :$ **skip**; $s \xrightarrow{s} M, \circ, s$ reduction.
- Successful runtime checks simplify s and leave $M'_\tau = M_\tau$ and $M'_o = M_o$. Thus $M', id \models s'$.

Lemma 11. *Other-thread steps preserve runtime type safety.* If

$$i \neq j \qquad \models M \qquad M, i \models s_i \qquad \text{checked}(M, E_i, i, s_i) \qquad M, E_i, i : s_i \to M', f, s'_i$$

then $M, j \models s_j \Rightarrow M', j \models s_j$
Proof: The only runtime typing judgment that could be affected by a step in thread i is ADDRESS-R. But by Lemma 11, $M, j \models a : \tau$ implies $M', j \models a : \tau$, so type safety is preserved.

A.8 Preserving Consistency

We prove in two lemmas that single-thread steps preserve environmental and memory consistency.

Lemma 12. *Single-thread steps preserve environmental consistency.* If

$$\models M \qquad M, i \models s \qquad M, E_i, i : s \to M', f, s'$$

then $E_j, j \models M \Rightarrow E_j, j \models M'$
Proof: Lemma 1 guarantees that clause 3 of $E_j, j \models M'$ holds. Clauses 1 and 2 are trivial for all steps except $s = a_1 := \text{scast}_\tau\ a_2$. In that case, by the assumption that variables are unaddressed, $v_2 = M_v(a_2) \neq E_j(x)$, so $M'(v_2)$ cannot violate clauses 1 or 2.

Lemma 13. *Single-thread steps preserve memory consistency.* If

$$E, id \models M \qquad M, id \models s \qquad \text{checked}(M, E, id, s) \qquad M, E, id : s \to M', f, s'$$

then $\models M \Rightarrow \models M'$.
Proof: We prove the various aspects of consistency independently, each time by case inspection of the operational semantics' steps.

First, we note that the "no more than one writer" and "no other readers than the writers" are preserved by all steps: the only non-trivial cases are the execution of *chkread* and *chkwrite*, and the preconditions of Figure 5 guarantee these properties are preserved.

Second, we consider type and owner consistency, again proceeding case by case.

- Trivial for non-assignment steps, as

  $$\forall a \in \text{dom}(M').M_\tau(a) = M'_\tau(a) \wedge M_v(b) = M'_v(b) \wedge M_o(a) = M'_o(a)$$

- $a :=$ null: setting $M'_v(a) = 0$ only weakens the type and owner consistency requirements.
- $a := n$: As $M, id \models s$, $M_\tau(a) = m\ \text{int} = M'_\tau(a)$. So setting $M'_v(a) = n$ does not affect the type and owner consistency requirements.
- $a := \text{new}_\tau$. Let $m = \max(\text{dom}(M))$. $M, id \models s$ implies

  $$M_\tau(a) = m\ \text{ref}\ \tau = M'_\tau(a)$$

  The only differences between M and M' are that $M'_v(a) = m + 1$ and $M'_v(m + 1) = 0$. Type and owner consistency for address $m + 1$ follows from $M'_v(m + 1) = 0$. The type consistency for address a follows from $M'_\tau(m + 1) = \tau$. If $m = \texttt{private}$, from $M, id \models a : m\ \text{ref}\ \tau$ we get $M_o(a) = id = M'_o(m + 1)$, so owner consistency for address a holds.
- $a_1 := a_2$. $M, id \models s$ implies

  $$M_\tau(a_1) = m_1\ \sigma = M'_\tau(a_1) \qquad M_\tau(a_2) = m_2\ \sigma = M'_\tau(a_2)$$

  The only difference between M and M' is the value of $b = M'_v(a_1)$. If $\sigma = \text{ref}\ \tau$ and $b \neq 0$, the memory consistency of M implies that $M_\tau(b) = \tau = M'_\tau(b)$, so types are consistent in M'.
  If $M'_\tau(a_1) = \texttt{private ref (private}\ \sigma')$, then

  $$M'_\tau(a_2) = m_2\ \text{ref (}\texttt{private}\ \sigma')$$

  As all types ultimately come from the program, $\vdash M'_\tau(a_2)$, so $m_2 = \texttt{private}$. Thus $M, id \models s$ and the owner consistency of M imply $id = M_o(a_1) = M_o(a_2) = M_o(b)$, so owners are consistent in M'.
- $a_1 := \text{scast}_{m_1\ \sigma}\ a_2$. Let $v_2 = M_v(a_2)$. $M, id \models s$ implies

  $$M_\tau(a_1) = m\ \text{ref}\ (m_1\ \sigma) \qquad M_\tau(a_2) = \texttt{private ref}\ (m_2\ \sigma)$$

  checked(M, E, id, s) implies $M, id \models \mathit{oneref}(v_2)$, i.e., $M(a_2)$ is the only reference to v_2. In M':
  - The type and owner consistency clauses hold for addresses a_1 and a_2 by construction.
  - All addresses $b \in \text{dom}(M) - \{a_1, a_2, v_2\}$ have consistent types and owners, as $M'_\tau(b) = m_b\ \text{ref}\ \tau_b \Rightarrow M'_v(b) \neq v_2$ and $\forall c \neq v_2.M'_\tau(c) = M_\tau(c) \wedge M'_o(c) = M_o(c)$.
  - For address v_2: Let $c = M_v(v_2)$. We show type and owner consistency for the non-trivial $\sigma = \text{ref}\ \tau'$, $c \neq 0$ case. From $\models M$, we know $M_\tau(c) = \tau' = M'_\tau(c)$. As $M'_\tau(v_2) = m_1\ \sigma$, type consistency holds at v_2.
    To show owner consistency at v_2, we note that

    $$M_\tau(v_2) \neq \texttt{private ref (dynamic}\ \sigma')$$

    as that is an illegal type. The only non-trivial owner consistency case is $m_1\ \sigma = \texttt{private ref (private}\ \sigma')$, in which case (by the exclusion above) $M_\tau(v_2) = m_1\ \sigma$ (i.e. the sharing mode is unchanged). Then

    $$M_\tau(a_2) = \texttt{private ref}\ (m_2\ \sigma)$$

    so rule ADDRESS-R and memory consistency imply $M_o(c) = id$. Thus the owner consistency clause holds at v_2.

A.9 Thread Creation and Destruction

These two lemmas show that thread creation and destruction preserve type safety, runtime checks and memory and environmental consistency.

Lemma 14. *Thread creation.* If

$$\models M \qquad E_i, i \models M \qquad M, i \models s_i \qquad \text{checked}(M, E_i, i, s_i) \qquad i \neq j \qquad \vdash P \Rightarrow P' \qquad f()\{\tau_1 x_1 \ldots \tau_n x_n; s_j\} \in P' \qquad M' = \mathit{extend}(M, j, \tau_1, \ldots, \tau_n) \qquad m = \max(\text{dom}(M)) \qquad E_j = G[x_1 \to m + 1, \ldots, x_n \to m + n]$$

then

$$\models M' \qquad E_i, i \models M' \qquad M', i \models s_i \qquad \text{checked}(M', E_i, i, s_i) \qquad E_j, j \models M' \qquad M', j \models s_j \qquad \text{checked}(M', E_j, j, s_j)$$

Proof: By Lemma 7 and the definition of *extend*.

Lemma 15. *Thread destruction.* If

$$\models M \qquad E_i, i \models M \qquad M, i \models s \qquad \text{checked}(M, E_i, i, s) \qquad i \neq j \qquad E_j, j \models M \qquad M' = \mathit{threadexit}(M, E_j, j)$$

then

$$\models M' \qquad E_i, i \models M' \qquad M', i \models s \qquad \text{checked}(M', E_i, i, s)$$

Proof: From $E_j, j \models M$, all addresses a such that $M'_v(a) \neq M_v(a)$ have $M_o(a) = j$. As types and owners, etc are otherwise unchanged in M', this implies $\models M', E, i \models M', M', i \models s$, checked$(M', E, i, s)$ (note in particular that stable$(M, i, a) \Rightarrow$ stable(M', i, a), $M, i \models \omega(a) \Rightarrow M', i \models \omega(a)$, $M, i \models a : \tau \Rightarrow M', i \models a : \tau$).

A.10 Soundness

Theorem 1. *Soundness.* Let P be a program, f a thread in P, and assume $\vdash P \Rightarrow P'$. Let $M, (E_1, s_1), \ldots, (E_n, s_n)$ be the state after any step of the execution of P' with initial thread f. Then

$$\models M \qquad E_i, i \models M \qquad M, i \models s_i \qquad \text{checked}(M, E_i, i, s)$$

Proof: By induction over the steps of the operational semantics. To satisfy the preconditions of Lemma 14 when creating the initial thread f, we assume that a special thread 0 with environment $E_0 = G$ exists with $s_0 =$ **done**. The initial memory M_0 has no runtime checks, no readers or writers, and zero values for all global variables, so $\models M_0$. By construction, $G, 0 \models M_0$, $M_0, 0 \models$ **done** and checked$(M_0, G, 0,$ **done**$)$.

There are three kinds of transitions to consider (initial thread startup can be handled as a special case of thread creation):

- Single-thread step:

$$\frac{M, E, i : s_i \rightarrow M', \circ, s'_i}{M, \ldots, (E_i, s_i), \ldots \rightarrow M', \ldots, (E'_i, s'_i), \ldots}$$

  By induction, for all threads j

$$\models M \qquad E_j, j \models M \qquad M, j \models s_j \qquad \text{checked}(M, E_j, j, s_j)$$

  By Lemma 13, we conclude that $\models M'$.
  By Lemma 12, we conclude that $\forall j. E_j \models M'$.
  By Lemma 10, we conclude that $M, i \models s'_i$.
  By Lemma 11, we conclude that $\forall j \neq i. M', j \models s_j$.
  By Lemma 8, we conclude that checked(M', E_i, i, s'_i).
  By Lemma 9, we conclude that $\forall j \neq i. \mathit{checked}(M', E_j, j, s_j)$.
- Thread creation: Induction holds from Lemma 14 and the same arguments as the single-thread step.
- Thread destruction: Induction holds from Lemma 15.

Theorem 2. *Safety of* `private` *accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a, then either $M_\tau(a) =$ `dynamic` σ or $M_o(a) = id$.
Proof: From Theorem 1 and ADDRESS-R's precondition.

Theorem 3. *Safety of* `dynamic` *accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id reads cell a from memory M, then

$$M_\tau(a) = \texttt{private}\ \sigma \vee (id \in M_R(a) \wedge M_W(a) - \{id\} = \emptyset)$$

If thread id writes to cell a in memory M, then

$$M_\tau(a) = \texttt{private}\ \sigma \vee (M_W(a) = \{id\} \wedge M_R(a) - \{id\} = \emptyset)$$

.
Proof: Consider a step of the operational semantics where statement s of thread id with environemnt E writes to address a of memory M, and $M_\tau(a) =$ `dynamic` σ. From Theorem 1, we can conclude that checked(M, E, id, s), hence $M, i \models \mathit{chkwrite}(a)$, so $M_W(a) = \{id\} \wedge M_R(a) - \{id\} = \emptyset$. The argument for reads is identical.