Formalizing and Checking Thread Refinement for Data-Race-Free Execution Models*

Daniel Poetzl and Daniel Kroening

University of Oxford

Abstract. When optimizing a thread in a concurrent program (either done manually or by the compiler), it must be guaranteed that the resulting thread is a refinement of the original thread. Most definitions of refinement are formulated in terms of valid syntactic transformations on the program code, or in terms of valid transformations on thread execution traces. We present a new theory formulated instead in terms of state transitions between synchronization operations. Our new method shows refinement in more cases and leads to more efficient and simpler procedures for refinement checking. We develop the theory for the SC-for-DRF execution model (using locks for synchronization), and show that its application in compiler testing yields speedups of on average more than two orders of magnitude compared to a previous approach.

1 Introduction

The refinement problem between threads appears in various contexts, such as the modular verification of concurrent programs, the validation of compiler optimization passes, or compiler testing. Informally, a thread T' is a refinement of a thread T if for all possible concurrent contexts $C = T_0 \parallel \ldots \parallel T_{n-1}$ (where $\parallel$ denotes parallel composition), the set of final states reachable by $T' \parallel C$ is a subset of the set of final states reachable by $T \parallel C$. We consider the problem as an instance of validating code optimization (either manual or by an optimizing compiler): the optimized thread must be a refinement of the original thread.

We focus on refinement in the "SC for DRF" execution model [1], i.e., programs behave sequentially consistent (SC) [6] if their SC executions are free of data races, and programs containing data races have undefined semantics. A program that contains data races could thus end up in any final state. Synchronization is provided via lock(l) and unlock(l) operations. The model is similar to, e.g., pthreads with its variety of lock operations such as `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

The definition of refinement given in the first paragraph is not directly useful for automated or manual reasoning, as it would require the enumeration of all possible concurrent contexts C. We thus develop a new theory that is based on comparing the state transitions of the original thread and the transformed thread between synchronization operations. We improve over existing work both in terms of *precision* and *efficiency*. First, our theory allows to show refinement

* Supported by ERC project 280053 and SRC task 2269.002.

in cases where others fail. For example, we also allow the reordering of shared memory accesses out of critical sections (under certain circumstances); a transformation that is unsupported by other theories. Second, we show that applying our new specification method in a compiler testing setting leads to large performance gains. We can check whether two thread execution traces match on average 210 X as fast as a previous approach by Morisset et al. [12].

The rest of the paper is organized as follows. Section 2 introduces our state-based refinement formulation and compares it to previous event-based approaches on a concrete example. Section 3 formalizes state-based refinement. Section 4 shows that our formulation is more precise in that it supports more compiler optimizations than current theories. Section 5 evaluates our theory in the context of a compiler testing application that involves checking thread execution traces. Section 6 surveys related work. Section 7 concludes.

2 State-Based vs. Event-Based Refinement

Current theories of refinement for language-level memory models (such as the Java Memory Model or SC-for-DRF) are phrased in terms of transformations on thread execution traces (see e.g. [2, 11, 12, 14, 15]). We refer to this notion of refinement as *event-based refinement*. The trace transformations are then lifted to transformations on the program code. Thread traces are sequences of memory events (reads or writes) and synchronization events (lock or unlock). The valid transformations are given as descriptions of which *reorderings*, *eliminations*, and *introductions* of memory events on a trace are allowed. Checking whether a trace t' is a correctly transformed version of a trace t then amounts to determining whether there is a sequence of valid transformations that turns trace t into trace t'. If each trace t' of T' is a transformed version of a trace t of T, it follows that T' is a refinement of T.

We show in the following that instead of describing refinement via a sequence of valid transformations on traces, switching to a theory based on state transitions provides several benefits. We refer to our new approach as *state-based refinement*. In essence, in the state-based approach, we only require that traces t' and t perform the same transformations on the shared state between corresponding synchronization operations, and that t does not allow for more data races than t. In the next section, we illustrate the difference between the two approaches on an example.

2.1 Example

Consider Figure 1, which gives an original thread T, a (correctly) transformed version T', and a concurrent context C in the form of another thread. The threads access shared variables x, y, z and local variables a, b. The context C outputs the value of variable z in the final state. By inspecting $T' \parallel C$ and $T \parallel C$ (assuming initial state $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$), we see that both combinations produce the same possible outputs (0 or 2). In fact, T' and T exhibit the same behavior in any concurrent context C for which $T \parallel C$ is data-race-free.

```
1  void thread_orig() {
2    int a, b;
3    lock(l);
4    x = 1;
5    x = 2;
6    unlock(l);
7    a = x;
8    b = y;
9    lock(l);
10   if (b == 0)
11     x = 0;
12   unlock(l);
13 }
```

(a) Original thread

```
1  void thread_trans() {
2    int a, b;
3    lock(l);
4    x = 2;
5    unlock(l);
6    b = y;
7    a = x;
8    lock(l);
9    if (b == 0)
10     x = 0;
11   b = y;
12   unlock(l);
13 }
```

(b) Transformed thread

```
1  void context() {
2    int a;
3    lock(l);
4    a = x;
5    z = a;
6    unlock(l);
7    join(thread_{orig|
8                 trans});
9    printf("%d\n", z);
10 }
```

(c) Context

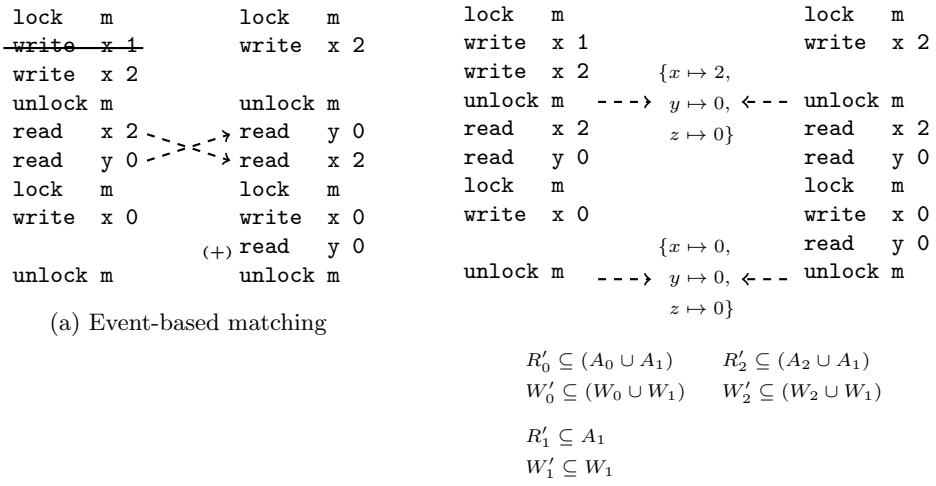
Fig. 1: Original thread T, transformed thread T', and concurrent context C

Now let us look at two traces t' of T' and t of T, and how a conventional event-based and our state-based theory would establish refinement. We assume for now that T and T' are only composed with contexts that do not write any shared memory locations accessed by them (as is the case for, e.g., the context given in Figure 1c). Figure 2 gives the execution traces of T (left trace) and T' (right trace) for initial state $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$.

A theory based on trace transformations (Figure 2a) would establish the refinement between the two traces by noting that `write x 1` can be removed ("overwritten write elimination"), `read x 2` and `read y 0` can be reordered ("non-conflicting read reordering"), and `read y 0` can be introduced ("irrelevant read introduction"). Showing refinement this way can become significantly more complicated and costly if longer traces and more optimizations are considered.

We specify trace refinement by requiring that t', t perform the same state transitions from lock to subsequent unlock operations, and that t' does not allow more data races than t. When assuming that the threads are only composed with contexts that do not write any shared memory locations, it is sufficient to check that t', t are in the *same state* at corresponding unlock operations. In this case, given an initial state s_{init}, we say a trace t is in state s at step i if s is like s_{init}, but updated with the values written by t up to step i. Indeed, both traces in Figure 2b are in state $\{x \mapsto 2, y \mapsto 0, z \mapsto 0\}$ at the first unlock(l), and in state $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$ at the second unlock(l). The *key reason* for why trace refinement can be specified this way is that any context C for which $T \parallel C$ is data-race-free can for each shared variable only observe the *last write* to it before an unlock operation. If it could observe any intermediate write, there would necessarily be a data race.

In addition to requiring that t' and t are in the same state, we also require that t' does not allow more data races than t. This requirement is captured by the set constraints in Figure 2b. The primed sets correspond to t', and the unprimed sets to t. The sets R'_i, R_i (W'_i, W_i) denote the sets of memory locations



(a) Event-based matching

(b) State-based matching

Fig. 2: Trace matching

read (written) between subsequent lock operations. For example, R_1 denotes the set of memory locations read by t between the first unlock(l) and the second lock(l). We also use the abbreviations $A'_i = R'_i \cup W'_i$ and $A_i = R_i \cup W_i$. As an example, the condition $W'_0 \subseteq W_0 \cup W_1$ says that any memory location written by t' between the first lock(l) and the subsequent unlock(l) must also be written by t either between the first lock(l) and the subsequent unlock(l), or between the first unlock(l) and the subsequent lock(l). Since for $x \in W'_0$ we require only that $x \in W_0$ or $x \in W_1$, this allows a write to move into the critical section in t' compared to t. We will define the set constraints more precisely in Section 3.

Contexts that Write In the case where a thread can be put in an arbitrary context that can also write to the shared state, when generating the traces we also need to take into account that a read of a variable x could yield a value that is both different from the initial value of x, and which the thread has not itself written (i.e., it was written by the context).

In an event-based theory this is typically handled by assuming that reads can return arbitrary values (see, e.g., [12]). However, this assumption is unnecessarily weak. For example, if a thread reads the same variable twice in a row with no intervening lock operation, and it did not itself write to the variable, then both reads need to return the same value. Otherwise, this would imply that another thread has written to the variable and thus there would be a data race.

In fact, when generating the traces of a thread, it is sufficient to assume that a thread observes the shared state only at its lock(l) operations. The reason for this is that lock(l) operations synchronize with preceding unlock(l) operations of other threads. And those threads in turn make their writes available at their unlock(l) operations.

3 Formalization

We now formalize the ideas from the previous section. For lack of space, we first make a few simplifying assumptions. Most notably we assume that threads do not contain nested locks (this assumption is lifted in the extended version of the paper [13]). We further assume that lock(l) and unlock(l) operations alternate on each thread execution, and that lock(l) and unlock(l) operations occur infinitely often on any infinite thread execution. This implies that a thread cannot get stuck, e.g., in an infinite loop without reaching a next lock operation. We also assume that the first operation in a thread is a lock(l), and the last *lock* operation is an unlock(l). We assume that the concurrent execution is the only source of nondeterminism, and that data races are the only source of undefined behavior.

3.1 Basics

A program $P = T_0 \| \ldots \| T_{n-1}$ is a parallel composition of threads $T_0, \ldots, T_{n-1}$. We denote by $h = (h_{T_0}, \ldots, h_{T_{n-1}})$ the vector of program counters of the threads. A program counter (pc) points at the next operation to be executed. We use the predicate $\mathsf{lock}(T, h)$ (resp. $\mathsf{unlock}(T, h)$) to denote that the next operation to be executed by thread T is a lock(l) (resp. unlock(l)). We use $\mathsf{term}(T, h)$ to denote that thread T has terminated.

Let M be a finite, fixed-size set of shared memory locations $x_1, \ldots, x_{|M|}$. A state is a total function $s\colon M \to V$ from M to the set of values V. We denote the set of all states by S. We assume there is a transition relation $\to$ between program configurations (P, h, s). We normally omit P when it is clear from context. The transition relation is generated according to interleaving semantics, and each transition step corresponds to an execution step of exactly one thread and accesses exactly one shared memory location or performs a lock operation. We denote by $h_s = (h_{s,T_0}, \ldots, h_{s,T_{n-1}})$ the initial pc vector with each thread at its entry point, and by $h_f = (h_{f,T_0}, \ldots, h_{f,T_{n-1}})$ the final pc vector with each thread having terminated.

We define a *program execution fragment* e as a (finite or infinite) sequence of configurations such that successive configurations are related by $\to$. A *program execution* is an execution fragment that starts in a configuration with pc vector h_s, and either has infinite length (i.e., does not terminate) or ends in a configuration with pc vector h_f. A *program execution prefix* is a finite-length execution fragment that starts in a configuration with pc vector h_s. Given an execution fragment such as $e = (h_0, s_0)(h_1, s_1) \ldots (h_n, s_n)$, we use indices 0 to $n-1$ to refer to the corresponding execution steps. For example, index 0 refers to the first execution step from (h_0, s_0) to (h_1, s_1).

We next define several predicates and functions on execution fragments (Figure 3). We usually omit the execution e when it is clear from context. The expression $\mathsf{src}(e, i)$ (resp. $\mathsf{tgt}(e, i)$) refers to the configuration to the left (resp. right) of $\to$ of the transition corresponding to step i of e.

We next define the semantics of a program according to interleaving semantics as the set of its initial/final state pairs.

$\mathsf{wr}(e,i)$:	step i is a shared write	$\mathsf{th}(e,i)$:	thread that performed step i
$\mathsf{rd}(e,i)$:	step i is a shared read	$\mathsf{src}(e,i)$:	source configuration of step i
$\mathsf{mem}(e,i)$:	$\mathsf{wr}(e,i) \vee \mathsf{rd}(e,i)$	$\mathsf{tgt}(e,i)$:	target configuration of step i
$\mathsf{lock}(e,i)$:	step i is a lock	$\mathsf{initial}(e)$:	initial state
$\mathsf{unlock}(e,i)$:	step i is an unlock	$\mathsf{final}(e)$:	final state of execution e, or $\bot$ if e is infinite
$\mathsf{loc}(e,i)$:	memory location/lock accessed by step i		

Fig. 3: Notation

Definition 1 (program semantics). $\mathbb{M}(P) = \{(s, s') \mid$ *there exists an execution* e *of* P *such that* $|e| < \infty \wedge \mathsf{initial}(e) = s \wedge \mathsf{final}(e) = s'\}$.

Only finite executions are relevant for the program semantics as defined above. Consequently, two programs P', P for which $\mathbb{M}(P') = \mathbb{M}(P)$ might have different behavior. For example, P' might have a nonterminating execution while P might always terminate. The programs P' and P are thus only *partially equivalent*.

We next define the relations sequenced-before (sb), synchronizes-with (sw), and happens-before (hb) for a given execution e (with $|e| = n$). It holds that $(i,j) \in \mathsf{sb}$ if $0 \leq i < j < n$ and $\mathsf{th}(i) = \mathsf{th}(j)$. It holds that $(i,j) \in \mathsf{sw}$ if $0 \leq i < j < n$, $\mathsf{unlock}(i)$, $\mathsf{lock}(j)$, and $\mathsf{loc}(i) = \mathsf{loc}(j)$. The happens-before relation hb is then the transitive closure of $\mathsf{sb} \cup \mathsf{sw}$.

Definition 2 (hb race). *We say an execution* e *(with* $|e| = n$*) contains an* hb data race*, written* $\mathsf{hb\text{-}race}(e)$*, if there are* $0 \leq i < j < n$ *such that* $\mathsf{th}(i) \neq \mathsf{th}(j)$*,* $\mathsf{loc}(i) = \mathsf{loc}(j)$*,* $\mathsf{wr}(i)$ *or* $\mathsf{wr}(j)$*, and* $(i,j) \notin \mathsf{hb}$.

We write $\mathsf{race}(P)$ to indicate that program P has an execution that contains an hb data race, and $\mathsf{race\text{-}free}(P)$ to indicate that it does not have an execution that has an hb data race. We are now in a position to define thread refinement.

Definition 3 (refinement). *We say that* T' *is a refinement of* T*, written* $\mathsf{ref}(T', T)$*, if the following holds:*

$$\forall\, C\colon \mathsf{race\text{-}free}(T \,\|\, C) \Rightarrow (\mathsf{race\text{-}free}(T' \,\|\, C) \wedge \mathbb{M}(T' \,\|\, C) \subseteq \mathbb{M}(T \,\|\, C))$$

The above defines that $\mathsf{ref}(T', T)$ holds when for all contexts C with which T is data-race-free, T' is also data-race-free, and the set of initial/final state pairs of $T' \,\|\, C$ is a subset of the set of initial/final state pairs of $T \,\|\, C$.

The above definition is not directly suited for automated refinement checking, as it would require implementing the $\forall$ quantifier (and hence enumerating all possible contexts C). We thus develop in the following our state-based refinement condition that implies $\mathsf{ref}(T', T)$, and which is more amenable to automated and manual reasoning about refinement.

3.2 State-Based Refinement

We next define the transition relation $\rightarrow_L$, which is more coarse-grained than $\rightarrow$. It will form the basis of the definition of our refinement condition.

Definition 4 ($\rightarrow_L$). $(P,h,s) \xrightarrow{l,(R_a,W_a),(R_b,W_b)}_L (P,h',s')$ *if and only if there exists an execution fragment* $e = (h_0,s_0)(h_1,s_1),\ldots,(h_k,s_k),\ldots,(h_n,s_n)$ *such that* $\mathsf{th}(0) = \mathsf{th}(1) = \ldots = \mathsf{th}(n-1) = T$ *for some thread* T *of* P, $\mathsf{lock}(0)$, $\mathsf{mem}(1),\ldots,\mathsf{mem}(k-1),\mathsf{unlock}(k)$, $\mathsf{mem}(k+1),\ldots,\mathsf{mem}(n-1)$, *either* $\mathsf{lock}(T, h_n)$ *or* $\mathsf{term}(T,h_n)$, $\mathit{loc}(0) = l$, $h_0 = h$ *and* $h_n = h'$. *The set* R_a *(resp.* W_a*) is the set of memory locations read (resp. written) by steps* 1 *to* $k-1$. *The set* R_b *(resp.* W_b*) is the set of locations read (resp. written) by steps* $k+1$ *to* $n-1$.

We also use the abbreviations $A_a = R_a \cup W_a$ and $A_b = R_b \cup W_b$. The relation $\rightarrow_L$ embodies uninterrupted execution of a thread T of P from a lock(l) to the next lock(l) (or the thread terminates). Since we have excluded nested locks, this means the thread executes exactly one unlock(l) in between. For example, in Figure 2b (left trace), the execution from the first lock in Line 1 to immediately before the second lock in Line 7 corresponds to a transition of $\rightarrow_L$. If we assume the thread starts in a state with all variables being 0, we have $s = \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$ and $s' = \{x \mapsto 2, y \mapsto 0, z \mapsto 0\}$. The corresponding access sets are $R_a = \{\}, W_a = \{x\}$, and $R_b = \{x,y\}, W_b = \{\}$.

We now define the semantics of a single thread T as the set of its *state traces*. A state trace is a finite sequence of the form $(l_0,s_0,R_0,W_0)(R_1,W_1,s_1)(l_2,s_2,R_2,W_2)(R_3,W_3,s_3)\ldots(l_{n-1},s_{n-1},R_{n-1},W_{n-1})(R_n,W_n,s_n)$. Two items $i, i{+}1$ (with i being even) of a state trace belong together. The item i corresponds to execution starting in state s_i at a lock(l) and executing up to the next unlock(l), with the thread reading the variables in R_i and writing the variables in W_i. The subsequent item $i+1$ corresponds to execution continuing at the unlock(l) and executing until the next lock(l) reaching state s_{i+1}, with the thread reading the variables in R_{i+1} and writing the variables in W_{i+1}.

The formal definition of the state trace set $\mathbb{S}(T)$ is given in Figure 4. Intuitively, the state trace set of a thread T embodies all interactions it could potentially have with a context C for which $\mathsf{race\text{-}free}(T \parallel C)$. A thread might observe writes by the context at a lock(l) operation. This is modeled in $\mathbb{S}(T)$ by the state changing between transitions. For example, the target state s_1 of the first transition is different from the source state s_2 of the second transition. The last line of the definition of $\mathbb{S}(T)$ constrains how the state may change between transitions. It defines that those memory locations that the thread T accesses in an execution portion from an unlock(l) to the next lock(l) (i.e., those in A_{i-1}) do not change at this lock(l). The reason for this is that if those memory locations would be written by the context, then there would be a data race. But since $\mathbb{S}(T)$ only models the potential interactions with race-free contexts, the last line excludes those state traces.

Previously we stated that we are interested in the states of a thread at lock and unlock operations, but $\mathbb{S}(T)$ embodies transitions from a lock(l) to the next lock(l). However, since we know the state at a lock(l), and we know the set of memory locations W_i written between the previous unlock(l) and that lock(l), we know the state of the memory locations $M - W_i$ at the unlock(l). This is sufficient for phrasing the refinement in the following.

We are now in a position to define the $\mathsf{match}_a(t',t)$ predicate. We will later extend it to the predicate $\mathsf{match}_b(t',t)$, which indicates whether a state trace

$$
\begin{aligned}
\mathbb{S}(T) = \{ & (l_0, s_0, R_0, W_0)(R_1, W_1, s_1)(l_2, s_2, R_2, W_2)(R_3, W_3, s_3) \dots (R_n, W_n, s_n) \mid \\
& \exists h_0, h_2, \dots, h_{n+1} : \\
& (T, h_0, s_0) \xrightarrow{l_0,(R_0,W_0),(R_1,W_1)}_L (T, h_2, s_1) \wedge \\
& (T, h_2, s_2) \xrightarrow{l_2,(R_2,W_2),(R_3,W_3)}_L (T, h_4, s_3) \wedge \\
& \dots \\
& (T, h_{n-1}, s_{n-1}) \xrightarrow{l_{n-1},(R_{n-1},W_{n-1}),(R_n,W_n)}_L (T, h_{n+1}, s_n) \wedge \\
& h_0 = h_s \wedge \\
& \forall i \in \mathsf{even}_n^+ : \forall x \in A_{i-1} : s_{i-1}(x) = s_i(x) \}
\end{aligned}
$$

Fig. 4: Definition of the state trace set of a thread

$t' \in \mathbb{S}(T')$ matches a state trace $t \in \mathbb{S}(T)$. The formal definition of $\mathsf{match}_a(t', t)$ is given in Figure 5. Primed symbols refer to components of t', and unprimed symbols refer to components of t. We denote by even_n (resp. odd_n) the set of all even (resp. odd) indices i such that $0 \leq i \leq n$. Intuitively, the constraints in Lines 3–6 specify that t' must not allow more data races than t. The constraints in Lines 3–4 correspond to an execution portion from a lock(l) to the next unlock(l), and Lines 5–6 correspond to an execution portion from the unlock(l) to the next lock(l). Since we have $R'_i \subseteq A_{i-1} \cup A_i \cup A_{i+1}$ and $W'_i \subseteq W_{i-1} \cup W_i \cup W_{i+1}$, the specification allows an access in t to move into a critical section in t' (we further investigate this in Section 4). The constraint in Line 7 specifies that t' and t receive the same new values at lock(l) operations (modeling writes by the context). The constraint at Line 9 specifies that the values written by t' and t before unlock(l) operations must be the same. The last constraint specifies that t' and t perform the same sequence of lock operations.

We next define the $\mathsf{match}_b(t', t)$ predicate. We denote by $t[0\!:\!i]$ the slice of a trace from index 0 to index i (exclusive).

Definition 5.

$$
\begin{aligned}
\mathsf{match}_b(T', T) \Leftrightarrow\ & \mathsf{match}_a(t', t) \vee \\
& \exists i \in \mathsf{even}^+ :\ \mathsf{match}_a(t'[0\!:\!i], t[0\!:\!i]) \wedge \\
& \qquad \exists x \in (A_{i-1} - A'_{i-1}) :\ s'_{i-1}(x) \neq s'_i(x)
\end{aligned}
$$

The above defines that either t' and t match, or there are same-length prefixes that match, and at the subsequent lock(l) a memory location in t' changes that is accessed by t but not by t' ($x \in A_{i-1} - A'_{i-1}$). Thus, a context that could perform the change of the memory location that t' observes would have a data race with t. Since when t is involved in a data race we have undefined behavior, any behavior of t' is allowed. Thus, t' and t are considered matched.

$\mathsf{match}_a(t', t) \Leftrightarrow$

1 $|t'| = |t|$
2 **let** $n = |t|$ **in**

\# `race constraints`
3 $\forall i \in \mathsf{even}_n \colon R'_i \subseteq (A_{i-1} \cup A_i \cup A_{i+1})$
4 $\forall i \in \mathsf{even}_n \colon W'_i \subseteq (W_{i-1} \cup W_i \cup W_{i+1})$
5 $\forall i \in \mathsf{odd}_n \colon R'_i \subseteq A_i$
6 $\forall i \in \mathsf{odd}_n \colon W'_i \subseteq W_i$

\# `state at locks constraints`
7 $\forall i \in \mathsf{even}_n \colon \forall x \in M - A_{i-1} \colon s'_i(x) = s_i(x)$
8 $\forall i \in \mathsf{even}_n \colon \forall x \in A_{i-1} - A'_{i-1} \colon s'_{i-1}(x) = s'_i(x)$

\# `state at unlocks constraints`
9 $\forall i \in \mathsf{odd}_n \colon \forall x \in M - W_i \colon s'_i(x) = s_i(x)$

\# `same locks constraint`
10 $\forall i \in \mathsf{even}_n \colon l'_i = l_i$

Fig. 5: Definition of matching state traces

We can now define our refinement specification $\mathsf{check}(T', T)$, which we later show implies the refinement specification $\mathsf{ref}(T', T)$ of Definition 3.

Definition 6 (check).

$$\mathsf{check}(T', T) \Leftrightarrow \forall t' \in \mathbb{S}(T') \colon \exists t \in \mathbb{S}(T) \colon \mathsf{match}_b(t', t)$$

We next state two lemmas that we use in the soundness proof of $\mathsf{check}(T', T)$. We refer to the extended version of the paper for the corresponding proofs [13].

Lemma 1 (coarse-grained interleaving). *Let e (with $|e| = n$) be an execution prefix of P with $\neg\mathsf{hb\text{-}race}(e)$ and $\mathsf{final}(e) = s$. Then there is an execution prefix e' of P with $\neg\mathsf{hb\text{-}race}(e')$ and $\mathsf{final}(e') = s$, such that execution portions from a $\mathrm{lock}(l)$ to the next $\mathrm{lock}(l)$ of a thread are not interleaved with other threads. Formally:*

$$\forall\, 0 \leq i < n \colon \mathsf{lock}(i) \Rightarrow \exists j > i \colon (\mathsf{lock}(\mathsf{th}(i), \mathsf{tgt}(j)) \vee \mathsf{term}(\mathsf{th}(i), \mathsf{tgt}(j)) \wedge \forall i < k < j \colon \mathsf{th}(k) = \mathsf{th}(i))$$

Lemma 2 (race refinement). *Let $\mathsf{check}(T', T)$. Then for all contexts C, if $T' \parallel C$ has an execution that has a data race, then $T \parallel C$ also has an execution that has a data race. Formally:*

$$\mathsf{check}(T', T) \Rightarrow \forall\, C \colon (\mathsf{race}(T' \parallel C) \Rightarrow \mathsf{race}(T \parallel C))$$

The following theorem establishes the soundness of our refinement condition $\mathsf{check}(T', T)$.

Theorem 1 (soundness). $\mathsf{check}(T', T) \Rightarrow \mathsf{ref}(T', T)$

Proof sketch. Let C be an arbitrary context C such that $\mathsf{race\text{-}free}(T \parallel C)$. Let further (s, s') in $\mathbb{M}(T' \parallel C)$. Thus, there is an execution e of $T' \parallel C$ that starts in state s and ends in state s'. By Lemma 2, $\mathsf{race\text{-}free}(T' \parallel C)$. Thus, by Lemma 1, there is an execution e' for which portions from a lock(l) to the next lock(l) of a thread are not interleaved with other threads. The sequence of those execution portions of T' corresponds to an element $t' \in \mathbb{S}(T')$. Then, by the definition of $\mathsf{check}(T', T)$, there is an element $t \in \mathbb{S}(T)$ such that either (a) $\mathsf{match}_a(t', t)$, or (b) $\exists i \in \mathsf{even}_n : \mathsf{match}_a(t'[0\colon i], t[0\colon i]) \wedge \exists x \in (A_{i-1} - A'_{i-1})\colon s'_{i-1}(x) \neq s'_i(x)$.

(a) Then t embodies the same state transitions as t'. This is ensured by constraints 7 and 9 of the definition of $\mathsf{match}_a()$. Constraint 7 specifies that the starting states of a transition match, and constraint 9 specifies that the resulting states of a transition match. A closer look at constraints 7 and 9 reveals that the corresponding states of t' and t do not need to be completely equal (only those memory locations in $M - A_{i-1}$ resp. $M - W_i$ need to have the same value). The reason for this is that if a thread would observe those memory locations it would give rise to a data race. Since we have both $\mathsf{race\text{-}free}(T' \parallel C)$ and $\mathsf{race\text{-}free}(T \parallel C)$, it follows that the values of the memory locations A_{i-1} resp. W_i can be arbitrary. Therefore, T can perform the same state transitions as T'. Thus, we can replace the steps of T' in e' by steps of T, and get a valid execution e'' of $T \parallel C$ ending in the same state. Therefore, $(s, s') \in \mathbb{M}(T \parallel C)$.

(b) Since $\mathsf{match}_a(t'[0\colon i], t[0\colon i])$, the first i state transitions of t are the same as those of t'. Thus, we can replace the first i execution portions of T' in e' by execution portions of T. The last execution portion of T accesses a memory location x that was not accessed by the corresponding execution portion of T' (since we have $\exists x \in A_{i-1} - A'_{i-1}$). Moreover, by $s'_{i-1}(x) \neq s'_i(x)$ it follows that this memory location is written by the context C. Thus, we have $\mathsf{race}(T \parallel C)$, which contradicts the premise $\mathsf{race\text{-}free}(T \parallel C)$. □

4 Supported Optimizations

We now investigate which optimizations are validated by our theory. By inspecting the definition of $\mathsf{match}_a()$ we see that it requires that t' and t perform the same state transitions between lock operations, and that the sets of memory locations accessed between lock operations of t' must be subsets of the corresponding sets of memory locations accessed by t. Together with the definitions of $\mathsf{match}_b()$ and $\mathsf{check}()$, this implies that if an optimization only performs transformations that do not change the state transitions between lock operations, and does not introduce accesses to new memory locations, then the optimized thread T' will be a refinement of the original thread T. This includes all the transformations shown to be sound by Boehm [2] and Morisset et al. [12] (considering programs using lock(l) and unlock(l) for synchronization).

```
1 lock(l);
2 x = 1;
3 y = 1;
4 unlock(l);
5 y = 2;
```

(a) Original (T)

```
1 lock(l);
2 x = 1;
3 y = 1;
4 y = 2;
5 unlock(l);
```

(b) Transformation 1 (T')

```
1 lock(l);
2 x = 1;
3 unlock(l);
4 y = 1;
5 y = 2;
```

(c) Transformation 2 (T'')

Fig. 6: Original, roach motel reordering, inverse roach motel reordering

Our theory also allows the reordering of shared memory accesses into and out of critical sections (under certain circumstances). The former are called *roach motel reorderings* and have been studied for example in the context of the Java memory model (see, e.g., [15]). The latter have not been previously described in the literature. In analogy to the former we term them *inverse roach motel reorderings*. We show on an example that our theory enables the proof of both optimizations.

Roach motel reorderings Consider Figure 6. Both x and y are shared variables. Figure 6a depicts the original thread T, and Figure 6b a correctly transformed version T'. The statement `y = 2` has been moved into the critical section. This is safe as it cannot introduce data races (but might remove data races).

Let t' be a state trace of T' starting in some initial state s_{init}. Then there is a state trace t of T starting also in s_{init}. The state s_{init} corresponds to the state at the first lock(l) for both threads. At the unlock(l) they are in states $s' = \{x \mapsto 1, y \mapsto 2\}$ resp. $s = \{x \mapsto 1, y \mapsto 1\}$. The access sets of the two state traces are $R'_0 = R'_1 = R_0 = R_1 = \{\}$ (we ignore the read sets in the following as they are empty), and $W'_0 = W_0 = \{x, y\}, W'_1 = \{\}, W_1 = \{y\}$. At the unlock($l$), according to the definition of $\mathsf{match}_a()$, the constraint $\forall x \in M - W_1 : s'(x) = s(x)$ needs to be satisfied. This is the case as the variable y for which s' and s differ is in W_1. Moreover, for $\mathsf{match}_a()$ to be satisfied, the following must hold for the write sets: $W'_0 \subseteq W_0 \cup W_1$ and $W'_1 \subseteq W_1$. This also holds. Hence, $\mathsf{match}_a(t', t)$ holds. Consequently, we also have $\mathsf{match}_b(t', t)$ and thus $\mathsf{check}(T', T)$, which implies $\mathsf{ref}(T', T)$ according to Theorem 1. Thread T' is thus a correctly transformed version of thread T.

Inverse roach motel reorderings Consider now the example in Figure 6c, which is a version T'' of the thread T. Again, it is correctly optimized. In order to get defined behavior for $T \parallel C$, the context C must in particular avoid data races with $\mathtt{y} = 2$. But this implies that the context cannot observe the write $\mathtt{y} = 1$, for if it could, there would be a data race with $\mathtt{y} = 2$. Moreover, moving $\mathtt{y} = 1$ downwards out of the critical section cannot introduce data races, as a write to y already occurs in this section. Consequently, $\mathtt{y} = 1$ can be moved downwards out of the critical section (or in this particular case removed completely).

We can use a similar argument as in the previous section to show within our theory that T'' is a correctly optimized version of T. Let t'', t be again two state traces starting in the same initial state s_{init}. At the unlock(l) they are in states $s'' = \{x \mapsto 1, y \mapsto y_{init}\}$ resp. $s = \{x \mapsto 1, y \mapsto 1\}$, with y_{init} denoting the value of y in s_{init}. Again, the constraints $\forall x \in M - W_1 \colon s''(x) = s(x)$, and $W_0'' \subseteq W_0 \cup W_1$ and $W_1'' \subseteq W_1$ are satisfied, and we can conclude that $\mathsf{match}_a(t'', t)$, $\mathsf{match}_b(t'', t)$, $\mathsf{check}(T'', T)$, and finally $\mathsf{ref}(T'', T)$ hold.

5 Evaluation

Previously we have argued that our specification efficiently captures thread refinement in the SC-for-DRF execution model, as it abstracts over the way in which a thread implements the state transitions between lock operations. In this section, we show that with our approach we can check in linear time whether two traces match. We also provide experimental data, showing that the application of our state-based approach in a compiler testing setting leads to large performance improvements compared to using an event-based approach.

5.1 Compiler Testing

Eide and Regehr [4] pioneered an approach to test that a compiler correctly optimizes programs that involves repeatedly (1) generating a random C program, (2) compiling it both with and without optimizations (e.g., `gcc -O0` and `gcc -O3`), (3) collecting a trace from both the original and the optimized program, and (4) checking whether the traces match. If two traces do not match, then a compiler bug has been found. Morisset et al. [12] extended this approach to a fragment of C11 and implemented it in their cmmtest tool.

The cmmtest tool consists of the following components: an adapted version of csmith [17] (we call it "csmith-sync" in the following) to generate random C threads, a tool to collect execution traces of a thread ("pin-interceptor"), and a tool to check whether two given traces match ("cmmtest-check"). The csmith-sync tool generates random C threads with synchronization operations such as `pthread_mutex_lock()`, `pthread_mutex_unlock()`, or the C11 primitives `release()` and `acquire()`. We only consider programs that contain lock operations. The pin-interceptor tool is based on the Pin binary instrumentation framework [10]. It executes a program and instruments the memory accesses and synchronization operations in order to collect a trace of those operations. The cmmtest-check tool takes two traces (produced by pin-interceptor) of an optimized and an unoptimized thread, and checks whether the traces match. We use the existing csmith-sync and pin-interceptor tools, and implemented our own trace checker tracecheck.

5.2 Complexity

Our tool tracecheck takes two traces (such as those depicted in Figure 2b), and first determines the states of the traces at lock operations, and the sets of memory

locations accessed between lock operations. That is, given a trace it constructs its corresponding state trace (i.e., an element of $\mathbb{S}(P)$). Then, it checks whether the two state traces match by evaluating the $\mathsf{match}_b()$ predicate. This way of checking traces is very efficient as it has runtime *linear* in the trace lengths.

This can be seen as follows. The size of a state is bounded by the number of writes that have occurred so far. Moreover, it is not necessary to check the complete states for equality at each lock operation; it suffices to check the memory locations that have been written to since the last check at the previous lock operation. Thus, checking the states at lock operations (corresponding to the "states at lock" and "states at unlock" constraints of the $\mathsf{match}_a()$ predicate) is a linear-time operation.

The race constraints can also be checked in linear time. First, the size of the sets is bounded by the number of memory locations accessed between the two corresponding lock operations. Second, subset checking between two sets A and B can be implemented in linear time.[1] In summary, we have a linear procedure for checking whether two traces match.

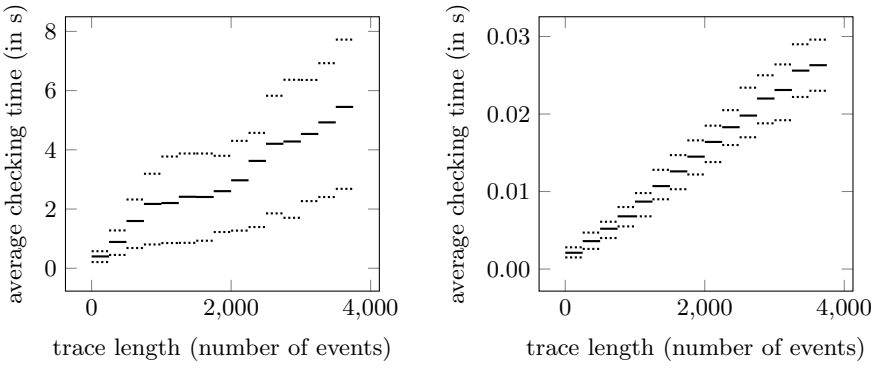
By contrast, cmmtest-check attempts to match traces by finding a sequence of valid transformations that transforms one trace into the other. Different sequences are explored in a tree-like fashion [12], suggesting exponential runtime in the worst case.

5.3 Experiments

We compared tracecheck to cmmtest-check on in total $40,000$ randomly generated C threads. We compiled each with `gcc -O0` and `gcc -O3` and collected a trace from each. The length of the traces was in the range of 1 to 4,000 events. We have chosen this range such that also cmmtest-check could match all the traces within the available memory limit. On some longer traces, cmmtest-check yields a stack overflow (it is implemented in the functional language OCaml). Our tool tracecheck can also handle traces with hundreds of thousands of events. Our tool outperformed cmmtest-check on all traces and was 210 X faster on average. Both tracecheck and cmmtest-check agreed on all traces, i.e., they either both classified a trace as correct or they both classified it as buggy.

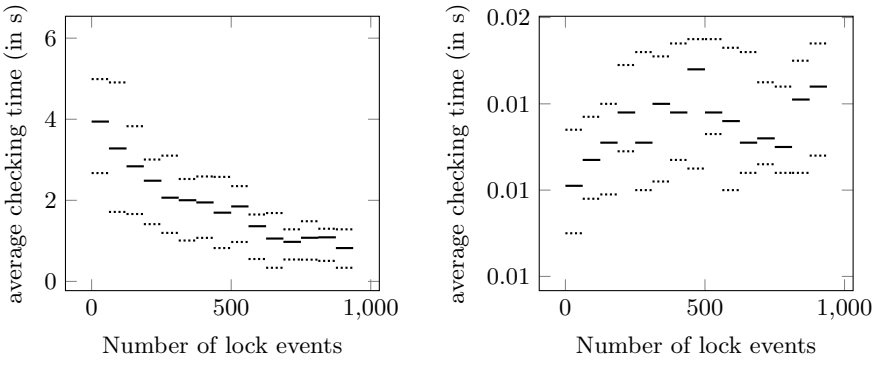
Figure 7 shows the average time it took to match two traces of a certain length, for cmmtest-check (Figure 7a) and tracecheck (Figure 7b). Along the x-axis, we classify the pairs of traces t', t into bins according to the length of the unoptimized trace t. Each bin i contains 100 pairs t', t such that the length of t is in the range $[250 \cdot i, 250 \cdot (i+1)]$. For example, bin 5 contains the pairs with the length of the unoptimized trace being in the range $[1250, 1500]$. The y-axis shows the average time it took to match two traces t', t in the respective bin. The dotted lines represent the 20th and 80th percentile to indicate the spread of the times.

[1] If A and B are represented as hash sets, then $A \subseteq B$ can be checked by iterating over the elements of A, and for each one performing a lookup in B (which has constant time). If all elements are found, A is a subset of B.



(a) cmmtest (b) tracecheck

Fig. 7: Average checking time over length of traces



(a) cmmtest (b) tracecheck

Fig. 8: Average checking time over number of locks in a trace

Figure 8 illustrates the effect of the number of lock operations in the two traces on the time it takes to check if they match. We have evaluated this on pairs of traces t', t with the unoptimized trace t having length in the range of $[1900, 2100]$. Along the x-axis, we classify the pairs of traces t', t into bins according to the number of lock operations they contain. The y-axis again indicates the average matching time. As can be seen in Figure 8a, cmmtest-check is sensitive to the number of locks in a trace. That is, matching traces generally takes longer the fewer locks they contain. The reason for this is that cmmtest-check considers lock operations as "barriers" against transformations: it does not try to reorder events across lock operations. Thus, the more lock operations there are in a trace, the fewer potential transformations it tries, and thus the lower the checking time. By contrast, the performance of our tool tracecheck is largely insensitive to the number of locks in a trace.

6 Related work

Refinement approaches can be classified based on whether they handle language-level memory models (such as SC-for-DRF or C11) [2, 11, 12, 14, 15], hardware memory models (such as TSO) [5, 16], or idealized models (typically SC) [3, 9].

The approaches for language-level models typically define refinement by giving valid transformations on thread execution traces. These trace transformations are then lifted to the program code level. An example is the theory of valid optimizations of Morisset et al. [12]. They handle the fragment of C11 with lock/unlock and release/acquire operations. The theory is relatively restrictive in that they do not allow the reordering of memory accesses across synchronization operations (such as the roach motel reorderings described in Section 4).

The approaches of Brookes [3] (for SC) and Jagadeesan [5] (for TSO) are closer to ours in that they also specify refinement in terms of state transitions rather than transformations on traces. They provide a sound and complete denotational specification of refinement. However, their completeness proofs rely on the addition of an unrealistic await() statement, which provides strong atomicity.

Liang et al. [7] presented a rely-guarantee-based approach to reason about thread refinement. Starting from the assumption of arbitrary concurrent contexts, they allow to add constraints that capture knowledge about the context in which the threads run in. They later extended their approach to also allow reasoning about whether the original and the refined thread exhibit the same termination behavior [8].

Lochbihler [9] provides a verified non-optimizing compiler for concurrent Java guaranteeing refinement between the threads in the source program and the bytecode. It is however based on SC semantics rather than the Java memory model. Sevcik et al. [16] developed the verified CompCertTSO compiler for compilation from a C-like language with TSO semantics to x86 assembly.

The compiler testing method based on checking traces of randomly generated programs on which we evaluated our refinement specification in Section 5 was pioneered by Eide and Regehr [4]. They used this approach to check the correct compilation of volatile variables. It was extended to a fragment of C11 by Morisset et al. [12].

7 Conclusions

We have presented a new theory of thread refinement for the SC-for-DRF execution model. The theory is based on matching the state of the transformed and the original thread at lock operations, and ensuring that the former does not introduce data races that were not possible with the latter. Our theory is more precise than previous ones in that it allows to show refinement in cases where others fail. It also boosts the efficiency of reasoning about refinement. Checking whether two traces match can be done in linear time, and consequently our implementation outperformed that of a previous approach by factor 210 X.

References

1. S. V. Adve and M. D. Hill. Weak ordering – a new definition. In *International Symposium on Computer Architecture (ISCA)*, pages 2–14. ACM, 1990.
2. H.-J. Boehm. Reordering constraints for Pthread-style locks. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 173–182. ACM, 2007.
3. S. Brookes. Full abstraction for a shared variable parallel language. In *Logic in Computer Science (LICS)*, pages 98–109. IEEE, 1993.
4. E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *Embedded Software (EMSOFT)*, pages 255–264. ACM, 2008.
5. R. Jagadeesan, G. Petri, and J. Riely. Brookes is relaxed, almost! In *Foundations of Software Science and Computational Structures (FoSSaCS)*, volume 7213 of *LNCS*, pages 180–194. Springer, 2012.
6. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *TC*, 100(9), 1979.
7. H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Principles of Programming Languages (POPL)*, pages 455–468. ACM, 2012.
8. H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Logic in Computer Science (LICS)*, pages 65:1–65:10. ACM, 2014.
9. A. Lochbihler. Verifying a compiler for Java threads. In *European Symposium on Programming (ESOP)*, volume 6012 of *LNCS*, pages 427–447. Springer, 2010.
10. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, pages 190–200. ACM, 2005.
11. J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, pages 378–391. ACM, 2005.
12. R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Programming Language Design and Implementation (PLDI)*, pages 187–196. ACM, 2013.
13. D. Poetzl and D. Kroening. Formalizing and checking thread refinement for data-race-free execution models (extended version). *CoRR*, abs/1505.08581, 2015.
14. J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *Programming Language Design and Implementation (PLDI)*, pages 306–316. ACM, 2011.
15. J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *Object-Oriented Programming, 22nd European Conference (ECOOP)*, volume 5142 of *LNCS*, pages 27–51. Springer, 2008.
16. J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *JACM*, 60(3), 2013.
17. X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)*, pages 283–294. ACM, 2011.