Safe Optimisations for Shared-Memory Concurrent Programs

Jaroslav Ševčík
University of Cambridge, The MathWorks
jarin.sevcik@gmail.com

Abstract

Current proposals for concurrent shared-memory languages, including C++ and C, provide sequential consistency only for programs without data races (the DRF guarantee). While the implications of such a contract for hardware optimisations are relatively well-understood, the correctness of compiler optimisations under the DRF guarantee is less clear, and experience with Java shows that this area is error-prone.

In this paper we give a rigorous study of optimisations that involve both reordering and elimination of memory reads and writes, covering many practically important optimisations. We first define powerful classes of transformations semantically, in a language-independent trace semantics. We prove that any composition of these transformations is sound with respect to the DRF guarantee, and moreover that they provide basic security guarantees (no thin-air reads) even for programs with data races. To give a concrete example, we apply our semantic results to a simple imperative language and prove that several syntactic transformations are safe for that language. We also discuss some surprising limitations of the DRF guarantee.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.3.4 [*Processors*]: Optimization; F.3.2 [*Semantics of Programming Languages*]

General Terms Languages, Reliability, Theory, Verification

Keywords Relaxed Memory Models, Semantics, Compiler Optimizations

1. Introduction

Standard compiler optimisation, such as common expression elimination, violate sequentially consistent semantics for multi-threaded programs. For example, observe that the following program cannot print value 1 in any of its interleavings.

initially `requestReady = responseReady = data = 0`

Thread 1	Thread 2

`data := 1`
`requestReady := 1`
`if (responseReady==1)`
`  print data`
	`if (requestReady==1) {`
	`  data := 2`
	`  responseReady := 1`
	`}`

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06...$10.00

However, an optimising compiler propagates the constant 1 from the `data:=1` statement and replaces[1] `print data` with `print 1`. While such an optimisation would be correct for sequential programs, it is not safe for this program in the sequentially consistent semantics because unlike the original program, the optimised program can output 1.

Such a result is worrisome because most theoreticians and practitioners assume a sequentially consistent semantics. Indeed, some researchers believe that compilers should perform only those optimisations that do not break the sequentially consistent semantics (see §7 for more details). In contrast, designers of common languages and hardware do permit and implement aggressive optimisations that can exhibit surprising behaviours for multi-threaded programs, but they provide mechanisms for recovering a sequentially consistent semantics. On multi-processors, programmers can constrain optimisations using memory fence instructions, which often have intricate semantics [3, 11, 20] and high run-time costs, usually in the order of tens to hundreds of cycles.

In higher-level programming languages, the recent trend is to guarantee interleaved semantics for programs without data races [6, 16]; we will call such a specification the *DRF guarantee* [1]. As programming without data races is considered good practice, the DRF guarantee is safe for well-engineered programs while validating most compilers because the possibly unintended effects of common optimisations cannot be observed without data races. However, to our knowledge, there is little rigorous evidence for this claim. We emphasise that the DRF guarantee leaves the behaviours of programs with races unspecified, possibly leading to executions where values appear "out-of-thin-air" [16]. This is unacceptable for languages that aim to give basic security guarantees for arbitrary programs, e.g., Java applets.

Contribution. We prove the DRF guarantee and the absence of out-of-thin-air values for a large class of compiler optimisations. In more detail, we design a novel trace-semantic characterisation of thread-local program transformations that is suitable for reasoning about validity of common optimisations of concurrent data race free programs. Using the characterisation, we establish that the transformations cannot introduce behaviours for race free programs and prevent out-of-thin-air values for arbitrary programs. We demonstrate the power of the semantic technique by applying it on several syntactic transformations, such as reorderings of independent statements, and eliminations of redundant memory accesses in the same block. The main advantage of the semantic approach is its independence from syntax: it allows using the same proof techniques for different languages, such as intermediate languages in compilers. We believe that our semantic transformations are general enough to capture most thread-local optimisations performed by realistic compilers. We also show that, perhaps surprisingly, re-

[1] For example, the `gcc` compiler version 4.1.2 on the x86 architecture performs this optimisation.

dundant read introduction invalidates some optimisations that are otherwise safe for the DRF-guarantee.

Approach. We view programs as sets of traces of its individual threads (§3); optimisations are modelled as relations on these tracesets (§4). Our main result shows that given a finite chain of programs, where the first program is data race free and the optimisation relation relates adjacent programs in the chain, the set of behaviours of the last program is a subset of the set of behaviours of the first program (§5), where the behaviours are sequences of externally observable actions (input or output) of all interleavings of the program.

To show the absence of out-of-thin-air values we observe that all our semantic transformations preserve an important property: if a thread writes or outputs a value then it must have read the value before. Using this property, we prove our out-of-thin-air guarantee: if a program does not contain constant c explicitly and there is no way to build c (for example because c is an integer and the program does not contain any arithmetic), then no transformation of the program can read, write or output value c (§5).

We have carried most of our work on semantic level, but we show how to apply our work by defining a simple imperative language with synchronisation primitives together with several simple syntactic transformations, and proving them safe (§6).

2. Traces, DRF and Transformations

In our examples, we use a simple C-like language. By convention, variables with the name beginning with `r` are local (registers), the remaining variables reside in distinct shared-memory locations. Similarly to Java [10, 16] or C++0x [4–6], some locations can be designated by the programmer as *volatile* (atomics in C++0x). We make the syntax and semantics formal in §6. Intuitively, volatile locations are intended for synchronisation between threads and for the purposes of the DRF guarantee, data races on volatile locations do not count as data races, i.e., a program is data race free if it cannot perform two conflicting accesses to the same non-volatile location at the same time. We give a formal definition of data race freedom in §3. Technically, the set of volatile locations should be part of a program. In our examples, all locations are non-volatile, unless stated otherwise. We assume that all locations are zero-initialised.

We represent programs as sets of memory action traces where the trace is a sequence of memory actions of a single thread. We have the following memory actions: $\mathrm{R}[\mathtt{l}{=}v]$ is a read from location l with value v; $\mathrm{W}[\mathtt{l}{=}v]$ a write to l with value v, $\mathrm{L}[\mathtt{m}]$ lock of monitor m; $\mathrm{U}[\mathtt{m}]$ an unlock of m; $\mathrm{X}(v)$ an external action (input or output) with value v; $\mathrm{S}(e)$ is a thread start action with entry point e, where the entry point is a thread identifier.

2.1 Transformations by Example

We consider four classes of program transformations and illustrate them on simple examples. We emphasise that the syntactic examples cover only small range of all the transformations allowed by the semantics.

Trace preserving transformations. Since our trace semantics only includes *shared-memory optimisations*, many otherwise non-trivial optimisations, such as loop unrolling or inlining, are identity optimisations in the trace semantics because they do not affect memory accesses. Interestingly, the trace semantics does not directly capture syntactic dependencies. For example, the code snippets `r:=x; if (r==0) y:=1 else y:=1` and `r:=x; y:=1` have the same sets of traces—the set of all sequences of a read of `x` followed by a write of 1 to `y`.

Thread 0	Thread 1
	`r1:=y`
`x:=2`	`print r1`
`y:=1`	`r1:=x`
`x:=1`	`r2:=x`
	`print r2`

(original)

Thread 0	Thread 1
	`r1:=y`
	`print r1`
`y:=1`	`r1:=x`
`x:=1`	`r2:=r1`
	`print r2`

(transformed)

Figure 1. Elimination example.

Elimination. The example in Fig. 1 shows an elimination of an overwritten write to `x` in the first thread and an elimination of a redundant read from `x` in the second thread. Note that such a transformation is not safe in a sequentially consistent semantics: unlike the original program, the transformed program can output 1 followed by 0, assuming that all memory locations are zero-initialised. This does not violate the DRF guarantee since the program contains data races on `x` and `y`. In the absence of data races, we show that eliminations cannot introduce new behaviours (§5).

Eliminations are easy to describe on traces: intuitively, a program is an elimination of another program if for each trace of the transformed program there is a trace in the original program such that we can obtain the transformed trace by eliminating some redundant actions from the original trace. For example, consider the trace $t = [\mathrm{S}(1), \mathrm{R}[\mathtt{y}{=}1], \mathrm{X}(1), \mathrm{R}[\mathtt{x}{=}0], \mathrm{X}(0)]$ of Thread 1 of the transformed program from Fig. 1 and observe that we can obtain t from trace $[\mathrm{S}(1), \mathrm{R}[\mathtt{y}{=}1], \mathrm{X}(1), \mathrm{R}[\mathtt{x}{=}0], \mathrm{R}[\mathtt{x}{=}0], \mathrm{X}(0)]$ of the original program by removing the redundant read of `x`. We consider the read redundant because there is an earlier read from the same location of the same value. We give the precise definition of various kinds of redundant actions and of the semantic elimination in §4. The semantic elimination transformation is general enough to cover optimisations that eliminate memory accesses based on data-flow analyses, i.e., common subexpression elimination, constant propagation, or even loop-invariant hoisting if combined with loop unrolling.

Reordering. In the example in Fig. 2, we show a reordering of a read from `y` with a later write to `x`. Again, this transformation is not safe in the interleaved semantics because the original program cannot print 1, as opposed to the transformed program (assuming zero-initialised memory). For semantic reordering we only require each trace of the transformed program to be a permutation of some trace of the original program with certain restrictions, such as preventing reordering of two conflicting accesses to the same memory location. Note that this definition can allow reordering of actions that are control dependent or falsely data dependent. Using our notion of reordering, the code snippet `r:=x; if (r==1) {y:=1;z:=1}else {z:=1;y:=1}` is a reordering of `y:=1;z:=1;r:=x` because any trace of the latter is a permutation of a trace of the former. We give a precise definition of the semantic reordering in §4 and show its safety for DRF programs in §5. The semantic reordering transformation covers code motion transformations, which are typically employed in loop optimisations.

Introduction. It is known that write introduction (sometimes called write speculation) generally violates the DRF guarantee because the introduced write might be seen by a different thread even though there was no data race in the original program [6]. It is less clear whether a redundant read introduction can violate the DRF guarantee in practice, especially in the seemingly harmless case where the program never uses the value obtained from the introduced read. Unsurprisingly, if we introduce irrelevant reads and execute on a sequentially consistent architecture, the reads cannot

```
Thread 0 || Thread 1          Thread 0 || Thread 1
         || r2:=y                      || x:=1
 r1:=x   || x:=1               r1:=x   || r2:=y
 y:=r1   || print r2           y:=r1   || print r2

    (original)                    (transformed)
```

Figure 2. Reordering example.

```
lock m      || lock m           r1 := y     || r2 := x
x := 1      || y := 1           lock m      || lock m
print y     || print x          x := 1      || y := 1
unlock m    || unlock m         print y     || print x
                                unlock m    || unlock m

    (a) original                (b) with introduced reads

r1 := y     || r2 := x
lock m      || lock m
x := 1      || y := 1
print r1    || print r2
unlock m    || unlock m

  (c) after read elimination
```

Can the program print two zeros?

Figure 3. Irrelevant read introduction.

change the behaviours. However, if combined with otherwise DRF-friendly optimisations, we can obtain non-sequentially consistent behaviour from programs that are data race free. For example, note that the first program from Fig. 3 cannot print two zeros, but if an optimiser inserts irrelevant reads and then reuses the introduced reads to eliminate other reads, as illustrated by the programs (b) and (c) in the figure, the resulting program can print two zeros even on a sequentially consistent architecture.

One might find both the optimisations from Fig. 3 dubious, but compilers (including `gcc`) do introduce reads when hoisting reads from a loop. Although we have not seen the redundant read elimination across synchronisation in any compiler yet, this optimisation has been proposed and implemented in `gcc` for the upcoming C++0x implementation [12]. In any case, this case of redundant read elimination is safe in the DRF guarantee (§4).

3. Trace Semantics

We begin the technical development with setting up our intuitive trace semantics rigorously.

Actions, Traces and Interleavings. Our traces are sequences of memory operations. In addition to the standard read, write, thread start and synchronisation operations, the traces also include external I/O operations, such as printing, because ultimately we wish to reason about observable I/O behaviours.

The thread start action is always the first action of a thread. Its purpose is to provide a connection between the identity of a thread and its entry point. To simplify the discussion, we create threads statically and we use thread identifiers as entry points.

We will use the following terminology to refer to classes of actions: a *memory access* to location l is a read or a write to l; a *volatile memory access* (resp. *read*, *write*) is a memory access (resp. read, write) to a volatile location; a *normal memory access* (resp. *read*, *write*) is an access (resp. read, write) to a non-volatile location; an *acquire* action is either a lock or a volatile read; a *release* action is an unlock or a volatile write; a *synchronisation* action is an acquire or release action.

To work with sequences of actions we use the following notation: $t +\!\!+ t'$ is a *concatenation* of lists t and t'; we write $t \leq t'$ if t is prefix of t', i.e., if there is s such that $t +\!\!+ s = t'$. Trace t is a *strict prefix* of t' ($t < t'$), if $t \leq t'$ and $t \neq t'$; $|t|$ denotes the length of the sequence t; t_i is i-th element of the list t, indices are 0-based; $[a \leftarrow t.\, P(a)]$ stands for the list of all actions in list t that satisfy condition P, in functional languages, this is often written as `filter` P t; we generalise the filter notation to a *map-filter* notation; the expression $[f(a) \mid a \leftarrow t.\, P(a)]$ denotes the list $[a \leftarrow t.\, P(a)]$ with each element transformed by function f, in functional languages, one would write this as `map` f (`filter` P t); $t|_S$ is a *sublist* of t that contains all elements with indices from S; for example, $[a, b, c, d]|_{\{1,3\}}$ is $[b, d]$; $\mathrm{dom}(t)$ is the set of all indices to t: $\mathrm{dom}(t) = \{0, \ldots, |t| - 1\}$; $\mathrm{ldom}(t)$ is the list of all indices to t in the increasing order: $\mathrm{ldom}(t) = [0, \ldots, |t| - 1]$.

Programs are represented as sets of traces, called tracesets. The traces in a traceset do not have to be complete; their execution can finish at any point. We model this by assuming that the set of traces of a program is prefix-closed, i.e., for traceset T, $t \leq t'$ and $t' \in T$ implies $t \in T$. We also require the tracesets to be *well locked*, i.e., for each $t \in T$ and monitor m, the number of unlocks of m in t is not greater than the number of locks of m in t. All traces in a traceset must be *properly started* meaning that if a trace is not empty its first action must be a start action.

For example, the traceset of the first program in Fig. 2 is the prefix closure of the following set (V is the set of values):

$$\{[\mathrm{S}(0), \mathrm{R}[\mathtt{x}{=}v], \mathrm{W}[\mathtt{y}{=}v]] \mid v \in V\}$$
$$\cup\ \{[\mathrm{S}(1), \mathrm{R}[\mathtt{y}{=}v], \mathrm{W}[\mathtt{x}{=}1], \mathrm{X}(v)] \mid v \in V\}.$$

We should note that this notion of traceset is rather weak as it does not enforce determinism or receptiveness. For instance, the set of traces $\{[\mathrm{S}(0)], [\mathrm{S}(0), \mathrm{R}[\mathtt{x}{=}1]], [\mathrm{S}(0), \mathrm{W}[\mathtt{y}{=}1]]\}$ is a valid traceset. Having non-determinism is useful to model underspecified features of languages, such as the loose evaluation order in C/C++.

Interleavings and Executions. Interleavings are sequences of thread-identifier–action pairs. For a pair $p = \langle \theta, a \rangle$, we write $\mathcal{A}(p)$ to refer to the action a, and $\mathcal{T}(p)$ for θ. A sequence of such pairs is an *interleaving*. Given an interleaving I, the *trace of θ in I* is the sequence of actions of thread θ in I, i.e., $[\mathcal{A}(p) \mid p \leftarrow I.\, \mathcal{T}(p) = \theta]$. In the text, we often omit the projection $\mathcal{A}(-)$ and write "I_i is a read" instead of "$\mathcal{A}(I_i)$ is a read". We may also omit the interleaving I and write "i is a read" if I is obvious from the context.

Interleaving is an execution of traceset T that respects mutual exclusion, its reads see the values of most recent writes and the traces of its thread are in T. Formally, interleaving I is an *interleaving of traceset T* if for all thread identifiers θ, the trace of θ is in T, thread identifiers correspond to entry-points, i.e., $\mathcal{A}(I_i) = \mathrm{S}(\theta)$ implies $\mathcal{T}(I_i) = \theta$ for all i and θ, and $\mathcal{A}(I_i) = \mathrm{L}[\mathtt{m}]$ implies that for each thread $\theta \neq \mathcal{T}(I_i)$ we have

$$|\{j \mid j < i \wedge \mathcal{T}(I_j) = \theta \wedge \mathcal{A}(I_j) = \mathrm{L}[\mathtt{m}]\}| =$$
$$|\{j \mid j < i \wedge \mathcal{T}(I_j) = \theta \wedge \mathcal{A}(I_j) = \mathrm{U}[\mathtt{m}]\}|.$$

We say that $r \in \mathrm{dom}(I)$ (i) *sees* write w if $\mathcal{A}(I_r) = \mathrm{R}[\mathtt{x}{=}v]$, $\mathcal{A}(I_w) = \mathrm{W}[\mathtt{x}{=}v]$ for some $\mathtt{x}$, v and $w < r$, and for all i such that $w < i < r$ the action I_i is not a write to l, (ii) *sees default value* if I_r is a read of the default value from l (typically 0) and there is no write $i < r$ to location l in I, (iii) *sees the most recent write* if r sees the default value or it sees some write w or r is not a read. Interleaving I is *sequentially consistent* if all $j \in \mathrm{dom}(I)$ see the most recent write in I. Sequentially consistent interleavings of T are called *executions* of T.

Orders on Actions. The sequencing of actions in interleavings imposes a total order on the execution of actions. In reality, actions are often performed concurrently. We will model this by constructing a partial happens-before order [13], which relates actions only if they are ordered by the program code or by synchronisation.

First, we define the program order to relate actions of the same thread in interleaving I, i.e., $\leq^I_{po} = \{(i,j) \mid 0 \leq i \leq j < |I| \wedge \mathcal{T}(I_i) = \mathcal{T}(I_j)\}$. We say that i *synchronises-with* j, written $i <^I_{sw} j$, if $i < j < |I|$ and $\mathcal{A}(I_i), \mathcal{A}(I_j)$ are a release-acquire pair, where actions a and b are a *release-acquire* pair if a is an unlock of monitor m and b is a lock of m, or a is a write to a volatile location l and b is a read of l. The *happens-before order* of I is the transitive closure of program order and synchronizes-with.

Note that $i \leq^I_{po} j$ implies $i \leq j$, and i synchronises-with j implies $i < j$. Hence, $i \leq^I_{hb} j$ implies $i \leq j$. As any subset of a total order is antisymmetric and $\leq^I_{hb}$ is transitive and reflexive by construction, the happens-before order is a partial order.

A matching is a function that relates the actions in two traces or interleavings. Formally, *matching* between lists I and I' is a partial injective function f from $\mathrm{dom}(I)$ to $\mathrm{dom}(I')$ such that $I_i = I'_{f(i)}$ for all $i \in \mathrm{dom}(f)$. The matching f is *complete* if $\mathrm{dom}(f) = \mathrm{dom}(I)$. We use matchings to relate actions in a trace (resp. interleaving) of a transformed program to a trace (resp. interleaving) of the original program.

Data Race Freedom. Two actions are *conflicting* if they access the same non-volatile location and at least one of them is a write. An interleaving *has a data race* if it contains two adjacent conflicting actions from different threads. A traceset is *data race free* if none of its executions has a data race. Equivalently, one could define data race freedom using the happens-before relation: a program is data race free if in all its executions, all the pairs of conflicting actions are ordered by the happens-before order of the execution [6, 21].

The common way of ensuring data race freedom is protecting every shared-memory location with a lock. Then there cannot be a data race because there must be an unlock-lock pair of actions on the same monitor between any two accesses to the same location in any execution. Alternatively, we can use volatile locations to make a program data race free. For example, if we mark the locations `requestReady` and `responseReady` in the first program in §1 as volatile, the program becomes data race free.

4. Semantic Transformations

We now define eliminations and reorderings semantically.

Eliminations. To define semantic read eliminations, we introduce *wildcard traces*. The wildcard traces are generalisations of ordinary traces, where each each element of a wildcard trace is either an action or a wildcard read $\mathrm{R}[\mathtt{x}{=}*]$. We use the wildcards to express independence of the trace's validity on the value that the wildcard reads might read. We say that a (normal) trace t is an *instance* of a wildcard trace t', if we can obtain t by replacing all wildcards in t' with some concrete values. A wildcard trace *belongs-to* traceset T if T contains all instances of the trace.

Similarly, we define wildcard interleavings to be interleavings with some ordinary actions replaced by wildcard reads. We obtain an *instance of a wildcard interleaving* by replacing each wildcard read by a read of the same location with the value of the most recent write to the same location, or with the default value if there is no earlier write to the same location. As opposed to trace instances, the instance of an interleaving is unique. We say that a wildcard interleaving *belongs-to* T if for each thread θ, the (wildcard) trace of θ belongs-to T.

For example, let T be the traceset of the following program

```
y:=1;       ||  r2:=y;
r1:=x;      ||  x:=1;
print r1;   ||
```

and observe that the wildcard traces $[\mathrm{S}(0), \mathrm{W}[\mathtt{y}{=}1], \mathrm{R}[\mathtt{x}{=}*]]$ and $[\mathrm{S}(1), \mathrm{R}[\mathtt{y}{=}*], \mathrm{W}[\mathtt{x}{=}1]]$ belong-to T. In contrast, the wildcard trace $[\mathrm{S}(0), \mathrm{W}[\mathtt{y}{=}1], \mathrm{R}[\mathtt{x}{=}*], \mathrm{X}(1)]$ does not belong-to T because some of its instances, e.g., $[\mathrm{S}(0), \mathrm{W}[\mathtt{y}{=}1], \mathrm{R}[\mathtt{x}{=}2], \mathrm{X}(1)]$, are not in T.

Our definition of eliminations on traces considers pairs of possibly non-adjacent memory accesses and identifies the conditions on the intervening actions that enable elimination of one of the accesses. Moreover, the definition allows removal of irrelevant (wildcard) reads and certain actions from the end of the trace. The "last-action" eliminations are useful for reordering.

Definition 1. *We say that there is a* release-acquire pair *between i and j in trace t if there are r and a such that $i < r < a < j$, t_r is a release and t_a is an acquire. Given trace t, we say that $i \in \mathrm{dom}(t)$ is*

1. redundant read after read *if $t_i = t_j = \mathrm{R}[\mathtt{l}{=}v]$ for some v, non-volatile $\mathtt{l}$ and $j < i$, and there is no release-acquire pair or write to $\mathtt{l}$ between j and i,*
2. redundant read after write *if $t_i = \mathrm{R}[\mathtt{l}{=}v]$, $t_j = \mathrm{W}[\mathtt{l}{=}v]$ for some v, non-volatile $\mathtt{l}$ and $j < i$, and there is no release-acquire pair or write to $\mathtt{l}$ between j and i,*
3. irrelevant read *if t_i is a wildcard non-volatile read,*
4. redundant write after read *if $t_i = \mathrm{W}[\mathtt{l}{=}v]$, $t_j = \mathrm{R}[\mathtt{l}{=}v]$ for some v, non-volatile $\mathtt{l}$ and $j < i$, and there is no release-acquire pair or other access to $\mathtt{l}$ between j and i,*
5. overwritten write *if $t_i = \mathrm{W}[\mathtt{l}{=}v]$, $t_j = \mathrm{W}[\mathtt{l}{=}v']$ for some v, v', non-volatile $\mathtt{l}$ and $j < i$, and there is no release-acquire pair or other access to $\mathtt{l}$ between j and i,*
6. redundant last write *if t_i is a normal write and there is no later release action and no later memory access to the same location,*
7. redundant release *if t_i is a release and there are no later synchronisation or external actions,*
8. redundant external action *if t_i is an external action and there are no later synchronisation or external actions.*

An index i is eliminable *in t if i satisfies one of the conditions above. Given traces t and t', the trace t' is an* elimination *of t if there is $S \subseteq \mathrm{dom}(t)$ such that $t' = t|_S$ and all $i \in \mathrm{dom}(t) \setminus S$ are eliminable in t. A traceset T' is an* elimination of a set of traces *T if each trace $t' \in T'$ is an elimination of some wildcard trace that belongs-to T.*

For example, in the wildcard trace

$$[\mathrm{S}(0), \mathrm{W}[\mathtt{x}{=}1], \mathrm{R}[\mathtt{y}{=}*], \mathrm{R}[\mathtt{x}{=}\mathbf{1}],$$
$$\mathrm{X}(1), \mathrm{L}[\mathtt{m}], \mathrm{W}[\mathtt{x}{=}\mathbf{2}], \mathrm{W}[\mathtt{x}{=}1], \mathrm{U}[\mathtt{m}]],$$

the indices 2, 3, and 6 are eliminable; so its elimination could be the trace $[\mathrm{S}(0), \mathrm{W}[\mathtt{x}{=}1], \mathrm{X}(1), \mathrm{L}[\mathtt{m}], \mathrm{W}[\mathtt{x}{=}1], \mathrm{U}[\mathtt{m}]]$. For an example of eliminations on tracesets, observe that all traces of the traceset of the program

```
x:=1; print 1; lock m; x:=1; unlock m;
```

are eliminations of some traces belonging-to the traceset of

```
x:=1; r1:=y; r2:=x; print r2;
if (r2!=0) {lock m; x:=2; x:= r2; unlock m;}
```

so the former traceset is an elimination of the latter.

Reordering. The reordering transformation allows changing order of execution of memory actions. However, not all permutations of actions preserve behaviours. We say that a is *reorderable* with b if either (i) a is a non-volatile memory access, and b is a non-conflicting non-volatile memory access, or an acquire action, or an external action; or (ii) b is a non-volatile memory access, and a is a non-conflicting non-volatile memory access, or a release, or an external action. The following table summarises the permissible reordering in a more readable form (cf. Doug Lea's cookbook **(author?)** [14]).

	$b =$				
$a =$	$\mathrm{W}[\mathtt{x}{=}v_x]^1$	$\mathrm{R}[\mathtt{x}{=}v_y]^1$	Acq	Rel	Ext
$\mathrm{W}[\mathtt{y}{=}v_y]^1$	$x \neq y$	$x \neq y$	✓	×	✓
$\mathrm{R}[\mathtt{y}{=}v_y]^1$	$x \neq y$	✓	✓	×	✓
Acquire	×	×	×	×	×
Release	✓	✓	×	×	×
External	✓	✓	×	×	×

Note that reorderability is not symmetric as we can reorder a write with a later acquire, but not the opposite. The only reason for the asymmetry is the so-called roach motel reordering [16], i.e., moving non-volatile memory accesses into synchronized blocks (for example, see rules R-RL, R-UW in Fig. 11).

A traceset T' is a reordering of a traceset T if each trace t' in T' is a permutation of some trace t from T. Moreover, the permutation has to satisfy two conditions: (i) it may only swap reorderable actions, (ii) if we apply the permutation to any prefix of t', i.e., if we leave out from t all the actions that are not in the prefix, then the resulting trace belongs to T.

In the rest of this section we will make this definition precise and then we apply the definition to a simple example. Given trace t, a bijection $f : \mathrm{dom}(t) \to \mathrm{dom}(t)$ is a *reordering function* for t if for all $i < j$ we have that $f(j) < f(i)$ implies that t_j is reorderable with t_i. It might seem that t_i should reorderable with t_j and not the opposite, but reordering function transforms traces in the opposite way: from traces of the transformed program to traces of the original program.

Before lifting the notion of reordering to traces and tracesets, we define a *de-permutation of a prefix* of a given trace t using function f. Formally, for $n \leq |t|$ and bijection f on $\mathrm{dom}(t)$, the *de-permutation of* t *of length* n, denoted by $f^{\rightarrow}_{<n}(t)$, is the trace

$$\left[t_{f^{-1}(i)} \mid i \leftarrow \mathrm{ldom}(t).\ f^{-1}(i) < n\right].$$

The *de-permutation* of t, written $f^{\rightarrow}(t)$, is the de-permutation of t of length $|t|$.

Note f is a complete matching between t and $f^{\rightarrow}(t)$. Now we lift the notion of reordering to tracesets: Given a set of traces T and trace t', function $f : \mathrm{dom}(t') \to \mathrm{dom}(t')$ *de-permutes* t' to T if f is a reordering function for t' and for any $n \leq |t'|$ we have $f^{\rightarrow}_{<n}(t) \in T$. A set of traces T' is a *reordering of* a set of traces T if for each $t' \in T'$ there is a function that de-permutes t' into T.

We will demonstrate the application of the reordering definition on the example from Fig. 2. The tracesets of the program on the left is the prefix closure of the set

$$\begin{aligned} T = &\{[\mathrm{S}(0), \mathrm{R}[\mathtt{x}{=}v], \mathrm{W}[\mathtt{y}{=}v]] \mid v \in \mathbb{N}\} \cup \\ &\{[\mathrm{S}(1), \mathrm{R}[\mathtt{y}{=}v], \mathrm{W}[\mathtt{x}{=}1], \mathrm{X}(v)] \mid v \in \mathbb{N}\}. \end{aligned}$$

The traceset of the transformed program is the prefix closure of

$$\begin{aligned} T' = &\{[\mathrm{S}(0), \mathrm{R}[\mathtt{x}{=}v], \mathrm{W}[\mathtt{y}{=}v]] \mid v \in \mathbb{N}\} \cup \\ &\{[\mathrm{S}(1), \mathrm{W}[\mathtt{x}{=}1], \mathrm{R}[\mathtt{y}{=}v], \mathrm{X}(v)] \mid v \in \mathbb{N}\}. \end{aligned}$$

Ideally, we would like to show that T' is a reordering of T, i.e., that for any trace $t' \in T'$ there is permutation function de-permuting t' to T. However, this is not the case, because none of the two permutations of the trace $[\mathrm{S}(0), \mathrm{W}[\mathtt{x}{=}1]]$ belongs to T. Therefore, traceset T' cannot be a reordering of T.

This is where the eliminations become useful: we can obtain the trace $[\mathrm{S}(0), \mathrm{W}[\mathtt{x}{=}1]]$ by eliminating the irrelevant read from $\mathtt{y}$ from the wildcard trace $[\mathrm{S}(0), \mathrm{R}[\mathtt{y}{=}*], \mathrm{W}[\mathtt{x}{=}1]]$, which belongs to T. More precisely, let $\hat{T} = T \cup [\mathrm{S}(0), \mathrm{W}[\mathtt{x}{=}1]]$ and note that $\hat{T}$ is an elimination of T. It remains to check that T' is a reordering of $\hat{T}$. Let us illustrate this on $t' = [\mathrm{S}(0), \mathrm{W}[\mathtt{x}{=}1], \mathrm{R}[\mathtt{y}{=}1], \mathrm{X}(1)]$. Let

$$f = \{\langle 0, 0\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle, \langle 3, 3\rangle\} \qquad t^n = f^{\rightarrow}_{<n}(t')$$

The meaning of t^n is the following: for $n = |t'|$, t^n is obtained from t' by applying the function f so that $t'_i = t^n_{f(i)}$ or, equivalently, $t'_{f^{-1}(i)} = t^n_i$ for all $i \in \mathrm{dom}(t')$. If $n < |t'|$, then we use the transformation only on the prefix of t' of length n. For illustration, see Figure 4. Since $t^n \in \hat{T}$ for any $n \leq |t'| = 4$, we satisfy the definition of reordering and f reorders $\hat{T}$ to t'. Similarly, for all other traces t' from T' there is a function that de-permutes t' to $\hat{T}$.

We should note that most of the complexity here is required only to cover the roach-motel reorderings, i.e., reordering with synchronisation. In the absence of roach-motel reordering, we could dispense with the de-permutations of prefixes.

5. Safety of Transformations

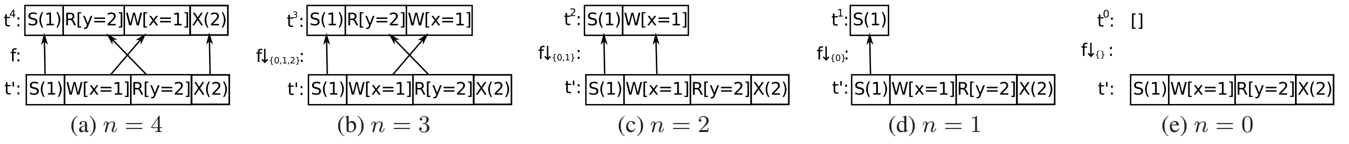
Here we sketch the main idea of our safety proof. The full details can be found in the author's PhD thesis [21]. We establish that both the elimination and reordering transformations have the following properties: (i) any execution of the transformed traceset has the same behaviour as some execution of the original traceset, provided that the original program was data race free; (ii) the transformations preserve data race freedom; (iii) the transformations cannot introduce values out-of-thin-air.

To prove (i) and (ii), we take an arbitrary execution of the transformed program and construct an execution of the original program that has the same behaviour. For both the elimination and the reordering transformations, we decompose the execution of the transformed program into traces for each thread, use the definitions of transformed tracesets from the previous section to obtain untransformed traces of the original traceset and then we compose the untransformed traces back into an untransformed interleaving so that the order of the external and synchronisation actions is preserved. Then we prove that either the constructed interleaving is an execution of the original traceset or there must have been a data race. Moreover, we show that the transformed program is data race free as the happens-before order of the constructed execution between two actions on the same variable implies happens-before ordering in the execution of the transformed program. We establish (iii) by showing that the transformations cannot introduce origins of values in tracesets, where a trace is an origin for v if there is a trace that contains a write of v or an output of v without any preceding read of v. The rest of this section describes the main proof ideas in more detail.

Elimination. We can prove sequential consistency for untransformed execution of an eliminated traceset directly because the untransformation of elimination embeds the happens-before order on each location. The main technical difficulty in the proof lies in showing that the extra actions introduced by the untransformation of elimination do not break sequential consistency.

Here we show that given a data race free traceset T, its elimination T', and an execution of T', we can untransform the execution so that the untransformation is an execution of T with the same behaviour as the execution of T'. The first step is the definition of the untransformation by lifting the definition of eliminations (Defi-

[1] Locations x and y are not volatile.



t⁴: S(1) R[y=2] W[x=1] X(2)
f:
t': S(1) W[x=1] R[y=2] X(2)
(a) $n = 4$
t³: S(1) R[y=2] W[x=1]
f↓{0,1,2}:
t': S(1) W[x=1] R[y=2] X(2)
(b) $n = 3$
t²: S(1) W[x=1]
f↓{0,1}:
t': S(1) W[x=1] R[y=2] X(2)
(c) $n = 2$
t¹: S(1)
f↓{0}:
t': S(1) W[x=1] R[y=2] X(2)
(d) $n = 1$
t⁰: []
f↓{}:
t': S(1) W[x=1] R[y=2] X(2)
(e) $n = 0$

Figure 4. Example of reordering traces.

nition 1) to interleavings. A naïve definition would require that the eliminated interleaving is just a sublist with some eliminable actions left out. However, this does not guarantee sequential consistency for volatile locations in the untransformed interleaving, because the untransformation might introduce a volatile write action. Instead we will allow the untransformation to swap some actions while preserving the program order and the order of synchronisation and external actions. Moreover, all the release and external actions introduced by the untransformation must be ordered after the release and external actions from the interleaving of the transformed program.

The precise definition follows. An *index i is eliminable in an interleaving I* if the corresponding index in the trace of $\mathcal{T}(I_i)$, i.e., the index $|\{j \mid j < i \wedge \mathcal{T}(I_i) = \mathcal{T}(I_j)\}|$, is eliminable in the trace of $\mathcal{T}(I_i)$ in I. Function f is an *unelimination* function from interleaving I' to wildcard interleaving I if f is a complete matching between I' and I such that (i) if $i < j \in \mathrm{dom}(I')$ and $\mathcal{T}(I'_i) = \mathcal{T}(I'_j)$ then $f(i) < f(j)$, (ii) if $i < j \in \mathrm{dom}(I')$ and $\mathcal{A}(I'_i)$, $\mathcal{A}(I'_j)$ are synchronisation or external actions then $f(i) < f(j)$, (iii) if $i \in \mathrm{rng}(f)$, $j \in \mathrm{dom}(I) \setminus \mathrm{rng}(f)$ and $\mathcal{A}(I_i)$, $\mathcal{A}(I_j)$ are synchronisation or external actions, then $i < j$, (iv) if $i \in \mathrm{dom}(I) \setminus \mathrm{rng}(f)$, then i is eliminable in I.

Lemma 1. *Let traceset T' be an elimination of traceset T and I' an interleaving of T'. Then there is a wildcard interleaving I belonging-to T and an unelimination function f from I' to I.*

We construct the function f and the uneliminated interleaving in three steps: we decompose the interleaving I' into individual threads, then we obtain 'uneliminated' traces for each thread, and finally we interleave the 'uneliminated' traces of the threads so that we preserve the order of synchronisation and external actions from I' while ordering all introduced synchronisation and external actions after the synchronisation and external actions from I'.

For example, consider the program (v is volatile)

```
v:=1;  ||  r1:=x;
y:=1;  ||  r2:=v;print r2;
```

By our definition of elimination on tracesets, we can eliminate the last release `v:=1` in the first thread and the irrelevant read `r1:=x` in the second thread:

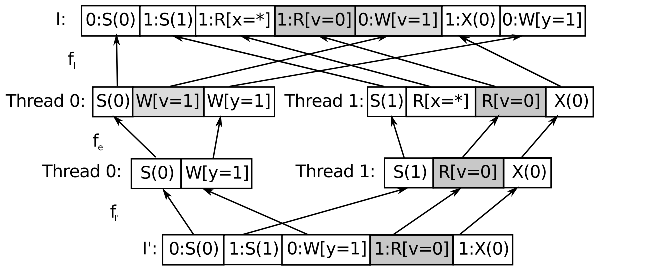
```
y:=1;  ||  r2:=v;print r2;
```

Consider the following execution of the program:

$$I' = [\langle 0, \mathrm{S}(0)\rangle, \langle 1, \mathrm{S}(1)\rangle, \langle 0, \mathrm{W}[\mathrm{y}{=}1]\rangle, \langle 1, \mathrm{R}[\mathrm{v}{=}0]\rangle, \langle 1, \mathrm{X}(0)\rangle]$$

Figure 5 shows one possible construction of unelimination I of I'. The unelimination function is a composition of the functions $f_{I'}$, f_e and f_I. For example, the unelimination function maps 2 to 6, i.e., it moves the second action of I' to the last position in I. Note that we cannot just insert the eliminated actions back into I' to get the unelimination because we would have to insert the write W[v=1] between the start of thread 0 and the write to y and this would break sequential consistency for the read of v.

Uneliminations have an important property: any unelimination of an execution is also an execution if the eliminated execution contained at most one data race. More precisely, let us have a data race free traceset T, its elimination T', an execution I' of T', a wildcard interleaving I belonging-to T and an unelimination function from I' to I, and let all strict prefixes of I' be data race free. Then the instance of I is an execution of T. Moreover, uneliminations preserve data races. These properties are not obvious. We refer the reader to **(author?)** [21] for a full proof. Consequently, elimination preserves data race freedom: suppose that an elimination of a data race free traceset was not data race free. Then we take the shortest execution I of the eliminated traceset with a data race. By the properties of unelimination, the uneliminated interleaving of I is an execution with a data race. This contradicts data race freedom of the original traceset. Thus, we conclude:



I: 0:S(0) 1:S(1) 1:R[x=*] 1:R[v=0] 0:W[v=1] 1:X(0) 0:W[y=1]
f_I
Thread 0: S(0) W[v=1] W[y=1]
Thread 1: S(1) R[x=*] R[v=0] X(0)
f_e
Thread 0: S(0) W[y=1]
Thread 1: S(1) R[v=0] X(0)
f_I'
I': 0:S(0) 1:S(1) 0:W[y=1] 1:R[v=0] 1:X(0)

Figure 5. Unelimination construction.

Theorem 1. *Let traceset T' be an elimination of a data free traceset T. Then T' is data race free and any execution of T' has the same behaviour as some execution of T.*

Reordering. Since the reordering untransformation does not preserve happens-before order in general, we cannot use the same direct proof we used for elimination. Instead, we prove the safety by induction on the size of the interleaving of the transformed program.

Just like with eliminations, we lift the notion of reordering to interleavings—we define an unordering function describing how to permute the actions in the transformed interleaving to get an interleaving of the original program. We require that whenever restricting an unordering on an interleaving to actions of one thread yields a reordering function on traces, as defined in §4.

Given traceset T and interleaving I', we say that complete matching $f : \mathrm{dom}(I') \to \mathrm{dom}(I')$ is an *unordering* from I' to T if we have: (i) if $i < j \in \mathrm{dom}(I')$, $\mathcal{T}(I'_i) = \mathcal{T}(I'_j)$ and $\mathcal{A}(I'_j), \mathcal{A}(I'_i)$ are not reorderable, then $f(i) < f(j)$, (ii) if $i < j \in \mathrm{dom}(I')$ and $\mathcal{A}(I'_i), \mathcal{A}(I'_j)$ are synchronisation or external actions, then $f(i) < f(j)$, (iii) for each thread θ, the permutation f restricted to actions of θ de-permutes the trace of θ in I' into T.

Using a similar construction to unelimination, unordering always exists. Unlike in the elimination safety proof, we establish the safety by induction on the size of the execution of the reordered traceset. To do that, we observe that restricting a reordering function for an execution to the prefix of the execution without the last element yields a valid reordering function. This allows us to prove by induction on the size of I' that for any unordering function f from an execution I' to a data race free traceset T, the interleaving $I = f^{\rightarrow}(I')$ is an execution of T. The technical details of this proof can be found in **(author?)** [21]. The DRF guarantee directly follows.

Theorem 2. *Suppose that traceset T' is a reordering of a data race free traceset T. Then any execution of T' has the same behaviour as some execution of T. Moreover, T' is data race free.*

The data race freedom of T' follows from reordering function being order-reflecting for happens-before order restricted to any individual memory location.

Out-of-thin-air. So far we have seen that the transformations provide an intuitive semantics for programs without data races. But what happens if there are data races in a program? Is anything possible? This would be unacceptable for languages that aim to give security guarantees for arbitrary programs, such as Java with sand-boxing. For an illustration of undesirable behaviours, consider the program

Initially, $\mathtt{x} = \mathtt{y} = 0$.

```
r1:=x;   ||  r2:=y;
y:=r1;   ||  x:=r2;
         ||  print r2;
```

Since the program does not contain value 42 nor any arithmetic that could create it, no transformation of the program should output 42.

Although our transformations do not give sequential consistency for programs with data races, we can show that out-of-thin-air behaviours, such as the one above, are impossible. More specifically, we will establish that for each output action of some value from an execution of a transformed program there is a statement in the original program that must have created that value. In a language without arithmetic, such as the one introduced in §6, this might mean that if a transformed program outputs value v, then v must be a default value, or the original program must have contained v in its program text as a constant. In a language with dynamic object allocation, we might use similar technique to show that if a program cannot allocate objects of a certain class in any thread, then in no transformed program the reference to such an object appears out-of-thin-air.

The guarantee is based on a simple observation: Let v be a value that is different from the default values. If a program without arithmetic does not contain v as a constant in the source code then in each trace, each write of the value v and each external action with the value v must be preceded by a read of the value v.

Formally, we say that trace t *is an origin* for value v if there is $i \in \mathrm{dom}(t)$ such that t_i is a write of v or an external action with the value v, and there is no $j < i$ such that t_j is a read of the value v. Later, in §6.1, we show an application of this semantic property for a simple language.

Lemma 2. *Let traceset T' be a reordering or an elimination of traceset T and suppose that no trace in T is an origin for v. Let us assume that no location has a singleton type with value v. Then no trace in T' is an origin for v.*

Finally, observe that if T does not contain an origin for a value, no execution of T can output that value:

Lemma 3. *Suppose that v is a value, that is not a default value for any type, T is a traceset, and no t in T is an origin for v. Then there is no execution of T that contains a read, write or external action with the value v.*

6. Connecting Syntax and Semantics

So far we have referred to an intuitive understanding of the relationship between programs and traces. To illustrate the transformations on a concrete syntax, we define a simple concurrent language and several simple but illustrative syntactic program transformations. Then we show that the syntactic transformations correspond to the semantic transformations and thus satisfy the DRF and out-of-thin-air guarantees. It is easy to add more language features, such as pointers, rich expression language and functions, without fundamental changes to the proofs because these features do not have any memory side-effects.

The syntax of our language is given in Fig. 6. The grammar uses distinct identifiers for thread-local register names, ranged over by r or $\mathtt{r1}, \mathtt{r2}$ in examples, natural numbers ranged over by i, location names (also called variables) ranged over by l, in examples $\mathtt{x}, \mathtt{y}, \mathtt{z}$, monitor names ranged over by m, in examples $\mathtt{m1}, \mathtt{m2}$.

We use a labellised small-step semantics to define the meaning of programs in the language introduced. A thread-local configuration is a triple $\langle \Lambda, \sigma, C\rangle$, where monitor state Λ is a function that maps monitor names to the nesting level of locks, local state σ maps register names to values, i.e., natural numbers, and C is a code fragment, which is either S or L or P from the syntax in Fig. 6. The only purpose of the monitor state is to prevent threads from issuing more unlocks than locks on each monitor.

The small-step relation $\langle \Lambda, \sigma, C\rangle \xrightarrow{a} \langle \Lambda', \sigma', C'\rangle$ takes a code fragment C in state Λ, σ to a code fragment C' and states Λ', σ' while issuing shared-memory action a. The action a may be empty, denoted by τ. Fig. 7 contains an inductive definition of the small step relation. We use the term $\mathrm{Val}(\sigma, E)$ for the value of the expression E in the environment σ, i.e., $\mathrm{Val}(\sigma, i) = i$ for any value $i \in \mathbb{N}$, $\mathrm{Val}(\sigma, r) = \sigma(r)$ for register name r, $\mathrm{Val}(\sigma, r_1\mathtt{==}r_2)$ (resp. $\mathrm{Val}(\sigma, r_1\mathtt{!=}r_2)$) is **tt** if $\mathrm{Val}(\sigma, r_1) = \mathrm{Val}(\sigma, r_2)$ (resp. $\mathrm{Val}(\sigma, r_1) = \mathrm{Val}(\sigma, r_2)$) or **ff** otherwise. We write $f[a \mapsto b]$ for a function update, i.e., $f[a \mapsto b](a) = b$ and $f[a \mapsto b](x) = f(x)$ for $x \neq a$.

We will write $\langle \Lambda, \sigma, C\rangle \underset{n}{\overset{t}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle$ for a sequence of n transitions, as defined in Fig. 8; notation $\langle \Lambda, \sigma, C\rangle \overset{t}{\Longrightarrow} \langle \Lambda', \sigma', C'\rangle$ stands for a finite number of transition, i.e., it is a shorthand for $\exists n.\ \langle \Lambda, \sigma, C\rangle \underset{n}{\overset{t}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle$. A configuration $\langle \Lambda, \sigma, C\rangle$ *may issue trace* t, written $\langle \Lambda, \sigma, C\rangle \Downarrow t$, if there are Λ', σ' and C' such that $\langle \Lambda, \sigma, C\rangle \overset{t}{\Longrightarrow} \langle \Lambda', \sigma', C'\rangle$. The meaning of a code fragment C in thread-local state Λ, σ is the set of all traces that it may issue, i.e., $[\![C]\!]_{\Lambda,\sigma} = \{t \mid \langle \Lambda, \sigma, C\rangle \Downarrow t\}$ The meaning of program P, written $[\![P]\!]$, is the set of traces $[\![P]\!]_{\Lambda_0,\sigma_0}$ where Λ_0 maps all monitor names to 0 and σ maps all locations to 0. Observe that $[\![P]\!]$ is a traceset.

6.1 Transformations: From Syntax to Semantics

Our basic template for local transformation t is given in Fig. 9. Observe that the rules for t cannot perform any transformations yet, i.e., for any P, we have $P \overset{t}{\rightsquigarrow} P$ (by induction on the structure of P) and if $P \overset{t}{\rightsquigarrow} P'$ then $P = P'$ (by induction on the derivation of $P \overset{t}{\rightsquigarrow} P'$). To allow some interesting transformations, we need to add some additional base rules.

Elimination. In Fig. 10, we give the additional base rules for our elimination transformation. Technically, the $\overset{e}{\rightsquigarrow}$ relation is defined inductively using the rules from Fig. 10 in addition to the rules from Fig. 9 with $\overset{t}{\rightsquigarrow}$ replaced by $\overset{e}{\rightsquigarrow}$. The elimination transformation removes redundant (shared-memory) reads and writes. The term $\mathrm{fv}(S)$ stands for all shared-memory locations contained in S. Statement S is sync-free if it does not contain any `lock` or `unlock` statements or accesses to volatile locations. Rule E-RAR (resp. E-RAW) removes redundant read if the value of the location is known from a previous read (resp. write). Rule E-WAR removes a write that follows a read of the same value in the same location. Overwritten writes can be eliminated by rule E-WBW. Rule E-IR

$$\begin{aligned} ri &::= \; r \;\mid\; i \\ T &::= \; ri \;\texttt{==}\; ri \;\mid\; ri \;\texttt{!=}\; ri \\ S &::= \; l \;\texttt{:=}\; r\texttt{;} \;\mid\; r \;\texttt{:=}\; l\texttt{;} \;\mid\; r \;\texttt{:=}\; ri\texttt{;} \;\mid\; \texttt{lock}\; m\texttt{;} \;\mid\; \texttt{unlock}\; m\texttt{;} \;\mid\; \texttt{skip;} \\ &\quad \mid\; \texttt{print}\; r\texttt{;} \;\mid\; \texttt{\{}L\texttt{\}} \;\mid\; \texttt{if (}T\texttt{)}\; S \;\texttt{else}\; S \;\mid\; \texttt{while (}T\texttt{)}\; S \\ L &::= \; S \;\mid\; S\,L \\ P &::= \; L \,||\, L \,||\, \dots \,||\, L \end{aligned}$$

Figure 6. A simple concurrent language – syntax.

$$\begin{array}{llll}
\langle \Lambda, \sigma, r\texttt{:=}ri\texttt{;}\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma[r \mapsto \mathrm{Val}(\sigma, ri)], \texttt{skip;}\rangle & & (\textsc{Regs}) \\
\langle \Lambda, \sigma, x\texttt{:=}r\texttt{;}\rangle & \xrightarrow{\mathrm{W}[x=\sigma(r)]} \langle \Lambda, \sigma, \texttt{skip;}\rangle & & (\textsc{Write}) \\
\langle \Lambda, \sigma, r\texttt{:=}x\texttt{;}\rangle & \xrightarrow{\mathrm{R}[x=v]} \langle \Lambda, \sigma[r \mapsto v], \texttt{skip;}\rangle & \text{where } v \in \tau(x) & (\textsc{Read}) \\
\langle \Lambda, \sigma, \texttt{lock}\; m\texttt{;}\rangle & \xrightarrow{\mathrm{L}[m]} \langle \Lambda[m \mapsto \Lambda(m)+1], \sigma, \texttt{skip;}\rangle & & (\textsc{Lock}) \\
\langle \Lambda, \sigma, \texttt{unlock}\; m\texttt{;}\rangle & \xrightarrow{\mathrm{U}[m]} \langle \Lambda[m \mapsto \Lambda(m)-1], \sigma, \texttt{skip;}\rangle & \text{where } \Lambda(m) > 0 & (\textsc{Ulk}) \\
\langle \Lambda, \sigma, \texttt{unlock}\; m\texttt{;}\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, \texttt{skip;}\rangle & \text{where } \Lambda(m) = 0 & (\textsc{E-Ulk}) \\
\langle \Lambda, \sigma, \texttt{print}\; r\texttt{;}\rangle & \xrightarrow{\mathrm{X}(\sigma(r))} \langle \Lambda, \sigma, \texttt{skip;}\rangle & & (\textsc{Ext}) \\
\langle \Lambda, \sigma, \texttt{if (}T\texttt{)}\; S_1 \;\texttt{else}\; S_2\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, S_1\rangle & \text{if } \mathrm{Val}(\sigma, T) = \mathbf{tt} & (\textsc{Cond-T}) \\
\langle \Lambda, \sigma, \texttt{if (}T\texttt{)}\; S_1 \;\texttt{else}\; S_2\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, S_2\rangle & \text{if } \mathrm{Val}(\sigma, T) = \mathbf{ff} & (\textsc{Cond-F}) \\
\langle \Lambda, \sigma, \texttt{while (}T\texttt{)}\; S\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, S\texttt{;while (}T\texttt{)}\; S\rangle & \text{if } \mathrm{Val}(\sigma, T) = \mathbf{tt} & (\textsc{Loop-T}) \\
\langle \Lambda, \sigma, \texttt{while (}T\texttt{)}\; S\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, \texttt{skip;}\rangle & \text{if } \mathrm{Val}(\sigma, T) = \mathbf{ff} & (\textsc{Loop-F}) \\
\langle \Lambda, \sigma, \texttt{skip;}L\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, L\rangle & & (\textsc{Seq}) \\
\langle \Lambda, \sigma, \texttt{\{skip;\}}\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, \texttt{skip;}\rangle & & (\textsc{Block}) \\
\langle \Lambda, \sigma, L_0 \,||\, \dots \,||\, L_n\rangle & \xrightarrow{\mathrm{S}(i)} \langle \Lambda, \sigma, L_i\rangle & \text{where } 0 \le i \le n & (\textsc{Par})
\end{array}$$

$$\frac{\langle \Lambda, \sigma, S\rangle \xrightarrow{a} \langle \Lambda', \sigma', S'\rangle}{\langle \Lambda, \sigma, S\;L\rangle \xrightarrow{a} \langle \Lambda', \sigma', S'L\rangle}(\textsc{Ev-Seq}) \qquad \frac{\Lambda, \sigma, L \xrightarrow{a} \Lambda', \sigma', L'}{\langle \Lambda, \sigma, \texttt{\{}L\texttt{\}}\rangle \xrightarrow{a} \langle \Lambda', \sigma', \texttt{\{}L'\texttt{\}}\rangle}(\textsc{Ev-Block})$$

Figure 7. Small-step Trace Semantics.

$$\frac{}{\langle \Lambda, \sigma, C\rangle \overset{[]}{\underset{0}{\Longrightarrow}} \langle \Lambda, \sigma, C\rangle}(\textsc{Tr-Id}) \qquad \frac{\langle \Lambda, \sigma, C\rangle \xrightarrow{\tau} \langle \Lambda'', \sigma'', C''\rangle \quad \langle \Lambda'', \sigma'', C''\rangle \overset{\alpha}{\underset{n}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle}{\langle \Lambda, \sigma, C\rangle \overset{\alpha}{\underset{n+1}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle}(\textsc{Tr-SeqT})$$

$$\frac{\langle \Lambda, \sigma, C\rangle \xrightarrow{a} \langle \Lambda'', \sigma'', C''\rangle \quad a \neq \tau \quad \langle \Lambda'', \sigma'', C''\rangle \overset{\alpha}{\underset{n}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle}{\langle \Lambda, \sigma, C\rangle \overset{a::\alpha}{\underset{n+1}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle}(\textsc{Tr-SeqA})$$

Figure 8. Multi-step Trace Semantics.

$$\frac{}{S \overset{t}{\rightsquigarrow} S}(\text{T-}\textsc{Id}) \qquad \frac{L \overset{t}{\rightsquigarrow} L'}{\texttt{\{}L\texttt{\}} \overset{t}{\rightsquigarrow} \texttt{\{}L'\texttt{\}}}(\text{T-}\textsc{Block}) \qquad \frac{S_1 \overset{t}{\rightsquigarrow} S_1' \quad L_2 \overset{t}{\rightsquigarrow} L_2'}{S_1\;L_2 \overset{t}{\rightsquigarrow} S_1'\;L_2'}(\text{T-}\textsc{Seq})$$

$$\frac{S_1 \overset{t}{\rightsquigarrow} S_1' \quad S_2 \overset{t}{\rightsquigarrow} S_2'}{\texttt{if (}T\texttt{) then}\; S_1 \;\texttt{else}\; S_2 \overset{t}{\rightsquigarrow} \texttt{if (}T\texttt{) then}\; S_1' \;\texttt{else}\; S_2'}(\text{T-}\textsc{If})$$

$$\frac{S \overset{t}{\rightsquigarrow} S'}{\texttt{while (}T\texttt{)}\; S \overset{t}{\rightsquigarrow} \texttt{while (}T\texttt{)}\; S'}(\text{T-}\textsc{while}) \qquad \frac{\forall i \in \{0, \dots, n\}.\; S_i \overset{t}{\rightsquigarrow} S_i'}{S_0 \,||\, \dots \,||\, S_n \overset{t}{\rightsquigarrow} S_0' \,||\, \dots \,||\, S_n'}(\text{T-}\textsc{Par})$$

Figure 9. Transformation template.

$$\frac{x \text{ not volatile} \quad r_1, r_2, x \notin \mathrm{fv}(S) \quad S \text{ sync-free}}{r_1\texttt{:=}x\texttt{;}\ S\texttt{;}\ r_2\texttt{:=}x \overset{e}{\leadsto} r_1\texttt{:=}x\texttt{;}\ S\texttt{;}\ r_2\texttt{:=}r_1}\ (\text{E-RAR}) \qquad \frac{x \text{ not volatile} \quad r_1, r_2, x \notin \mathrm{fv}(S) \quad S \text{ sync-free}}{x\texttt{:=}r_1\texttt{;}\ S\texttt{;}\ r_2\texttt{:=}x \overset{e}{\leadsto} x\texttt{:=}r_1\texttt{;}\ S\texttt{;}\ r_2\texttt{:=}r_1}\ (\text{E-RAW})$$

$$\frac{x \text{ not volatile} \quad r, x \notin \mathrm{fv}(S) \quad S \text{ sync-free}}{r\texttt{:=}x\texttt{;}\ S\texttt{;}\ x\texttt{:=}r \overset{e}{\leadsto} r\texttt{:=}x\texttt{;}\ S\texttt{;}}\ (\text{E-WAR}) \qquad \frac{x \text{ not volatile} \quad r_1, r_2, x \notin \mathrm{fv}(S) \quad S \text{ sync-free}}{x\texttt{:=}r_1\texttt{;}\ S\texttt{;}\ x\texttt{:=}r_2 \overset{e}{\leadsto} S\texttt{;}\ x\texttt{:=}r_2}\ (\text{E-WBW})$$

$$\frac{x \text{ not volatile}}{r\texttt{:=}x\texttt{;}\ r\texttt{:=}i \overset{e}{\leadsto} r\texttt{:=}i}\ (\text{E-IR})$$

Figure 10. Additional rules for syntactic elimination.

removes irrelevant reads which cannot affect the rest of the program because their value is thrown away.

Our definition of syntactic eliminations (Figure 10) does not include last action eliminations because they are not composable in the sense that trace t'_1 being a (last action) elimination of t_1 and t'_2 being an elimination of t_2 does not necessarily imply that $t'_1 + t'_2$ is an elimination of $t_1 + t_2$. To recover compositionality, we will call index i *properly eliminable* in a wildcard trace t if i is a redundant read after read, or a redundant read after write, or an irrelevant read, or a redundant write after read or an overwritten write. Given traces t and t', the trace t' is a *proper elimination* of t if there is S such that $t' = t|_S$ and all $i \in \mathrm{dom}(t) \setminus S$ are properly eliminable in t. We denote the proper elimination by $t \rightarrow_e t'$.

The following lemma clarifies the relationship between the syntactic elimination and the semantic elimination from §4.

Lemma 4. *Let C be a code fragment and $C \overset{e}{\leadsto} C'$. Then for any monitor states Λ, Λ', register states σ, σ' and trace t' we have:*

- *If $\langle \Lambda, \sigma, C' \rangle \Downarrow t'$ then there is a wildcard trace t such that $t \rightarrow_e t'$ and for any instance $\hat{t}$ of t we have $\langle \Lambda, \sigma, C \rangle \Downarrow \hat{t}$.*
- *If $\langle \Lambda, \sigma, C' \rangle \overset{t'}{\Longrightarrow} \langle \Lambda', \sigma', \texttt{skip;} \rangle$ then there is a wildcard trace t such that $t \rightarrow_e t'$ and for any instance $\hat{t}$ of t we have $\langle \Lambda, \sigma, C \rangle \overset{\hat{t}}{\Longrightarrow} \langle \Lambda', \sigma', \texttt{skip;} \rangle$.*

As a consequence, if $P \overset{e}{\leadsto} P'$ for some programs P and P', then $[\![P']\!]$ is an elimination of $[\![P]\!]$. Combining this observation with the results for the semantic elimination (Theorem 1) gives us a compositional DRF guarantee for syntactic elimination:

Theorem 3. *Suppose that $P \overset{e}{\leadsto} P'$ and $[\![P]\!]$ is data race free. Then $[\![P']\!]$ is data race free, and any execution of $[\![P']\!]$ has the same behaviour as some execution of $[\![P]\!]$.*

Reordering. Fig. 11 shows additional rules for reordering. Similarly to the elimination transformations, the relation $\overset{r}{\leadsto}$ is defined inductively using the rules from Fig. 11 and relabelled rules from Fig. 9. We capture reordering of independent non-volatile memory accesses in rules R-RW, R-RR, R-WR and R-WW. Rules R-WL, R-RL, R-UW and R-UR allow moving non-volatile memory accesses inside synchronised blocks. Note that the R-RR, R-WR and R-WW rules allow limited reordering of accesses to volatile locations with non-volatile accesses.

The relationship between the syntactic and semantic reordering is more involved in our framework because syntactic reordering corresponds to semantic elimination followed by semantic reordering; for an illustration, see the example in §4. The following lemma formalises the relationship between the syntactic reordering and the trace semantics.

Lemma 5. *Assume that $C \overset{r}{\leadsto} C'$. Then for each Λ and σ there is a prefix closed set of traces T satisfying these conditions: (i) the set of traces $[\![C]\!]_{\Lambda,\sigma}$ is a subset of T, (ii) each trace from T is an elimination of some wildcard trace that belongs-to $[\![C]\!]_{\Lambda,\sigma}$, (iii) for each trace t', if $\langle \Lambda, \sigma, C' \rangle \Downarrow t'$ holds then there is a function that de-permutes t' into T, (iv) for each trace t', if there are Λ' and σ' such that $\langle \Lambda, \sigma, C' \rangle \overset{t'}{\Longrightarrow} \langle \Lambda', \sigma', \texttt{skip;} \rangle$ then there is a function f that de-permutes t' into T and $\langle \Lambda, \sigma, C \rangle \overset{f^{\rightarrow}(t')}{\Longrightarrow} \langle \Lambda', \sigma', \texttt{skip;} \rangle$.*

Together with the semantic DRF guarantees from §5, we obtain the compositional DRF guarantee for syntactic reordering.

Theorem 4. *Suppose that $P \overset{r}{\leadsto} P'$ and $[\![P]\!]$ is data race free. Then $[\![P']\!]$ is data race free, and any execution of $[\![P']\!]$ has the same behaviour as some execution of $[\![P]\!]$.*

Out-of-thin-air. To establish the out-of-thin-air-guarantee for our simple language, we first observe that if a program does not contain constant c then the program is not an origin for c:

Lemma 6. *Let v be a value such that v is not a default value for any location, i.e., $v \neq 0$. Let P be a program without any statement of the form $r\texttt{:=}v$, where r is a register name. Then no trace in the traceset of P is an origin for the value v.*

This observation allows us to use our semantic observations from §5 and state the syntactic counterpart of the out-of-thin-air guarantee.

Theorem 5. *Suppose that c is a constant different from 0, and P a program that does not contain a statement of the form $r := c$, where r is a register. Let P' be a program obtained from P by any composition of syntactic reorderings or eliminations. Then P' cannot output c.*

7. Related Work

The existing research on compiler optimisations for shared-memory concurrency concentrates mainly on maintaining sequential consistency for all programs, starting with the foundational work of Shasha and Snir **(author?)** [24], where they take a sequentially consistent execution of a straight line program and describe a set of reordering constraints that preserve sequential consistency. Building on these foundations, papers **(author?)** [15, 17, 26] describe whole program analyses that determine allowable reorderings in multi-threaded programs and compilers that preserve behaviours of arbitrary programs. The emphasis of that line of work is different from ours: while they design a restricted compiler that guarantees (an illusion of) sequential consistency for all programs, we show that commonly used program transformations maintain an illusion of sequential consistency for correctly synchronised programs. We are not aware of any such work in the context of compilers.

Correctness of optimisations is closely related to weak memory models for programming languages and hardware. In fact, validity of common optimisations was the main motivation for designing the Java Memory Model [16]. Despite this, Java does not allow

$$\frac{r_1 \neq r_2 \quad x \text{ not volatile}}{r_1\texttt{:=}x\texttt{;}\ r_2\texttt{:=}y\texttt{;}\ \overset{r}{\rightsquigarrow}\ r_2\texttt{:=}y\texttt{;}\ r_1\texttt{:=}x\texttt{;}}\,(\text{R-RR})$$

$$\frac{x \neq y \quad y \text{ not volatile}}{x\texttt{:=}r_1\texttt{;}\ y\texttt{:=}r_2\texttt{;}\ \overset{r}{\rightsquigarrow}\ y\texttt{:=}r_2\texttt{;}\ x\texttt{:=}r_1\texttt{;}}\,(\text{R-WW})$$

$$\frac{r_1 \neq r_2 \quad x \neq y \quad x \text{ or } y \text{ not volatile}}{x\texttt{:=}r_1\texttt{;}\ r_2\texttt{:=}y\texttt{;}\ \overset{r}{\rightsquigarrow}\ r_2\texttt{:=}y\texttt{;}\ x\texttt{:=}r_1\texttt{;}}\,(\text{R-WR})$$

$$\frac{r_1 \neq r_2 \quad x \neq y \quad x, y \text{ not volatile}}{r_1\texttt{:=}x\texttt{;}\ y\texttt{:=}r_2\texttt{;}\ \overset{r}{\rightsquigarrow}\ y\texttt{:=}r_2\texttt{;}\ r_1\texttt{:=}x\texttt{;;}}\,(\text{R-RW})$$

$$\frac{x \text{ not volatile}}{x\texttt{:=}r\texttt{;}\ \texttt{lock}\ m\texttt{;}\ \overset{r}{\rightsquigarrow}\ \texttt{lock}\ m\texttt{;}\ x\texttt{:=}r\texttt{;}}\,(\text{R-WL})$$

$$\frac{x \text{ not volatile}}{r\texttt{:=}x\texttt{;}\ \texttt{lock}\ m\texttt{;}\ \overset{r}{\rightsquigarrow}\ \texttt{lock}\ m\texttt{;}\ r\texttt{:=}x\texttt{;}}\,(\text{R-RL})$$

$$\frac{x \text{ not volatile}}{\texttt{unlock}\ m\texttt{;}x\texttt{:=}r\texttt{;}\ \overset{r}{\rightsquigarrow}\ x\texttt{:=}r\texttt{;unlock}\ m\texttt{;}}\,(\text{R-UW})$$

$$\frac{x \text{ not volatile}}{\texttt{unlock}\ m\texttt{;}r\texttt{:=}x\texttt{;}\ \overset{r}{\rightsquigarrow}\ r\texttt{:=}x\texttt{;unlock}\ m\texttt{;}}\,(\text{R-UR})$$

$$\frac{r_1 \neq r_2 \quad x \text{ not volatile}}{\texttt{print}\ r_1\texttt{;}r_2\texttt{:=}x\texttt{;}\ \overset{r}{\leftrightsquigarrow}\ r_2\texttt{:=}x\texttt{;print}\ r_1\texttt{;}}\,(\text{R-XR})$$

$$\frac{x \text{ not volatile}}{\texttt{print}\ r_1\texttt{;}x\texttt{:=}r_2\texttt{;}\ \overset{r}{\leftrightsquigarrow}\ x\texttt{:=}r_2\texttt{;print}\ r_1\texttt{;}}\,(\text{R-XW})$$

Figure 11. Additional rules for syntactic reordering.

several common optimisations [9, 23], and even the reference implementation of Java Virtual Machine does not conform with the specification [22]. The situation does not seem to be much better in Microsoft Common Language Infrastructure (CLI): the official specification is in informal prose without any clear guarantees for programmers. In practice, the CLI virtual machine seems to be very conservative, and it appears that it conforms to a much stronger unofficial model **(author?)** [18]. There have been several other suggestions for weak memory models for high-level languages, but none of these addressed validity of compiler transformations [9, 19]. Hardware memory models or hardware-inspired memory models, such as **(author?)** [2, 7, 8], do not seem to be suitable for higher languages because they are either too prohibitive and do not allow reordering of reads with later independent writes/reads at all, or they rely on syntactic notions of dependency for reordering, which can be often removed by optimising compilers. We hope that our work will serve as a basis for a formal weak memory model defined in terms of permissible transformations.

8. Conclusion

We have proved that two large classes of compiler optimisations, elimination and reordering, are safe in the DRF guarantee, i.e., they cannot introduce new behaviours for data race free programs. Since our transformations preserve data race freedom, arbitrary composition of the transformations is also safe.

As reasoning about multi-threading is notoriously error-prone, we mechanised the definitions and difficult parts of the semantic elimination safety proof (such as Lemma 1) in the Isabelle/HOL proof assistant. The mechanisation has about ten thousand lines of proof script. The current development snapshot is available at `http://www.cl.cam.ac.uk/~js861/transafety/mm-traces`.

In our work, we assume that the transformed program always runs sequentially consistently. This might seem to be a severe limitation because few modern processors guarantee sequential consistency. However, it is well-understood how to ensure the DRF guarantee on hardware and all our transformations preserve data race freedom, so we can safely assume sequential consistency in hardware. Unfortunately, we cannot easily recover the out-of-thin-air guarantee without deeper understanding of the processor memory models. Therefore, we started investigating extensions of our techniques to cover hardware memory models. Our first results are encouraging—we can explain the Sun TSO memory model [25], used by most SPARC processors with our semantic transformations. We believe that there similar results can be achieved for other processor memory models.

Acknowledgments

I would like to thank David Aspinall for his guidance and Peter Sewell for many useful comments on earlier drafts of the paper. I acknowledge funding from EPSRC grants EP/F036345, EP/H005633.

References

[1] S. V. Adve and M. D. Hill. Weak ordering — a new definition. In *ISCA'90*, pages 2–14, 1990.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[3] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[5] P. Becker, editor. *Programming Languages — C++. Final Committee Draft.* 2010. ISO/IEC JTC1 SC22 WG21 N3092.

[6] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08*, pages 68–78, New York, NY, USA, 2008. ACM.

[7] Gérard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *POPL '09*, pages 392–403, 2009.

[8] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *CC '10*, pages 104–123, 2010.

[9] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *16th ESOP*, 2007.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series).* Addison-Wesley Professional, July 2005.

[11] Intel. A formal specification of Intel Itanium processor family memory ordering, 2002. Available from `http://www.intel.com/design/itanium/downloads/251429.htm`.

[12] P. Joisha, R. Schreiber, P. Banerjee, H.-J. Boehm, and D. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. Technical Report HPL-2010-81, HP Laboratories, 2010. To appear in POPL 2011.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[14] Doug Lea. The JSR-133 cookbook for compiler writers, 2008. `http://g.oswego.edu/dl/jmm/cookbook.html`.

[15] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *PPOPP*, pages 1–12. ACM, 1999.

[16] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *POPL '05*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[17] Samuel P. Midkiff and David A. Padua. Issues in the optimization of parallel programs. In *ICPP (2)*, pages 105–113. Pennsylvania State University Press, 1990.

[18] Vance Morrison. Understand the impact of low-lock techniques in multithreaded apps. *MSDN Magazine*, Oct 2005.

[19] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *PPoPP '07*. ACM, Mar 2007.

[20] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL'09*, January 2009.

[21] J. Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, Laboratory for Foundations of Computer Science, 2008.

[22] J. Ševčík. The Sun Hotspot JVM does not conform with the Java memory model. Technical Report EDI-INF-RR-1252, School of Informatics, University of Edinburgh, 2008.

[23] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP '08*, pages 27–51, 2008.

[24] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[25] Sparc International. Sparc architecture manual, version 9, 2000. Available from `http://developers.sun.com/solaris/articles/sparcv9.html`.

[26] Z. Sura, X. Fang, C. Wong, S. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP '05*, pages 2–13, New York, NY, USA, 2005. ACM.