A Theory of Memory Models (Extended Abstract) *

Vijay Saraswat
IBM TJ Watson Research Center
vijay@saraswat.org

Radha Jagadeesan
DePaul University
rjagadeesan@cs.depaul.edu

Maged Michael
IBM TJ Watson Research Center
magedm@us.ibm.com

Christoph von Praun
IBM TJ Watson Research Center
praun@us.ibm.com

Abstract

A *memory model* for a concurrent imperative programming language specifies which writes to shared variables may be seen by reads performed by other threads. We present a simple mathematical framework for relaxed memory models for programming languages. To instantiate this framework for a specific language, the designer must choose the notion of *atomic steps* supported by the language (e.g. 32-bit reads and writes) and specify how a composite step may be broken into a sequence of atomic steps (the *decomposition rule*). This rule determines which sequence of intermediate writes (if any) are visible to concurrent reads by other threads. Different choices of the rule lead to models which permit a read to return any value if there is a concurrent write (race), or models which satisfy a "No Thin Air Read" property. The former is suitable for languages such as C++ (programs with races have undefined behavior), and the latter for Java. Other intermediate models are possible, useful and interesting.

We establish that all models in the framework satisfy the *Fundamental Property* of relaxed memory models: programs whose sequentially consistent (SC) executions have no races must have have only SC executions. We show how to define synchronization constructs (such as volatiles of various kinds) in the framework, and discuss the causality test cases from the Java Memory Model.

Categories and Subject Descriptors D.3.1 [*Memory Models*]: Formal Definitions and Theory

General Terms Performance, Languages, Theory

Keywords Memory Model, Sequential Consistency, Weak Models, RAO

1. Introduction

Memory models address a central question of imperative concurrency: When can a write done by one thread be read by another?

Leslie Lamport provided a simple answer in [8]. Assume the state of the memory can be described by an assignment of values to variables. Assume that exactly one thread is permitted to perform exactly one read or write operation in a single step. Then the possible executions of the program are given by all possible interleavings of the steps of the threads making up the program. This notion of execution is called *Sequential Consistency (SC)*.

Unfortunately, SC is costly to implement. Weak processor architectures such as the Power PC require expensive barrier operations to implement SC. Further, SC is not consistent with a wide array of compiler optimizations geared towards optimizing the performance of single-threaded code. Such optimizations often work by rearranging the code of a single thread while guaranteeing that its *input/output (i/o) behavior* is unchanged. For any piece of sequential code s, let us define its *i/o function* $io(s)$ to be the function from total stores (mappings that assign a value to every variable) to total stores given by executing s in the input store and returning the final store. Consider, for instance, the program $P = \mathtt{x} = 1; \mathtt{r} = \mathtt{y}$. An implementation may replace this code by P': `r=y; x=1` since $io(P) = io(P')$. However, under SC these two code fragments are not identical. Consider running it in parallel with Q: `r0=x; if(r0==1)y=1`. Assume execution is initiated in a store in which `x, y=0, 0`. Now $P \mid Q$ may result in `r=1`, whereas $P' \mid Q$ will never do so.

It should be noted that Shasha and Snir [15] recognized this problem and proposed solutions involving extra computational overhead (e.g. the use of memory barriers/fences). There has been more recent work [16] on compiler analyses to reduce or eliminate the overhead of implementing SC. At this point we cannot definitively conclude that the overhead can be eliminated for a large class of programs. Therefore the need to define memory models is real.

1.1 Race-free programs

An important observation underlies nearly all research in this area. Consider again the program $P' \mid Q$ above. Let us say that steps executed by a program are related with a transitive, irreflexive partial order, the *happens-before* (*hb*) order [8]. One should interpret *p hb q* as saying that the step *p* must happen "before" the step *q* in any execution; i.e. *q* must observe the store in a state in which *p* has been performed. For instance, it is reasonable to require that all the steps taken by a single thread are totally ordered by *hb*, and synchronization operations (e.g. lock/unlock) must be used to (dynamically) introduce *hb* edges between steps of one thread and steps of another. Now since $P' \mid Q$ does not contain any synchronization operation, it has a *data race*: a thread (Q) has a step s (`r0=x`) that reads a variable (`x`) that another thread (P') writes in a step t (`x=1`)

* A fuller version of this paper can be fetched at http://www.saraswat.org/rao.html

PPoPP'07 March 14–17, 2007, San Jose, California, USA.

without there being an *hb*-edge from t to s.[1] If a program has no races then a thread T_1 does not read the value of a variable written into by another thread T_2 (without using a synchronization operation). Therefore T_1 will be insensitive to code reordering in T_2. Hence one can have one's cake (SC semantics) and eat it too (good performance).

Therefore it seems reasonable to require the *Fundamental Property*:

> Programs whose SC executions have no races must have only SC executions.

This raises the question: Who is responsible for ensuring that a program is race free ... the implementation[2] or the user?

It is plausible that the implementation should have this responsibility. Race analysis is a difficult technical problem and in some cases it may be permissible to incur the overhead of runtime detection of races. This approach is being pursued by some researchers. The general drawback is that it is hard to design static conditions that are general enough to recognize that arbitrary clever programs are race-free.

Indeed, it is often the case that a programmer—aware of the designed control flow of the program—can establish that a particular program is race-free based on global analysis. Therefore it seems plausible that the programmer should shoulder the responsibility of establishing the global property that synchronization-free access to shared variables will not lead to race conditions.

In return, the implementation should guarantee performance: it should be able to perform *all single thread optimizations* as long as they are consistent with explicit synchronization operations introduced by the programmer (if any). That is, the language should specify—and the implementation should realize—as *weak* semantics as possible for concurrent, unsynchronized read/writes to the same location (performing as many code reorderings as possible). Memory barriers should be introduced only as required by the semantics of synchronization operations in the language.

How weak is "weak"?

Fundamental Property. For programs **without** races, the Fundamental Property places a lower bound on behaviors and appears to be a reasonable "firewall". Most programmers may program in a world in which they write complete, race-free programs. Hence they need to reason only about SC executions.

Recently, Doug Lea has argued that volatiles should be permitted to satisfy a weaker requirement, *cache-coherent causal consistency* (CCCC, [9]). Volatiles are discussed further in Section 4.1. In essence volatile variables are intended to enable threads to reliably communicate values to each other via single reads and writes. Volatile variables specify certain synchronization conditions which ensure that there are no races in the sense in which we have defined them. Therefore programs in which all shared variables are volatile will, by definition, be race free. Doug argues that such programs should satisfy CCCC rather than SC. CCCC require consistency properties to hold only *locally*, that is, between pairs of processors communicating through reads/writes to the same memory location, and not globally (across all processors). Thus CCCC does not require unrestricted transitivity of the happens before relation. In particular CCCC permits different processors to have different (inconsistent) views about the ordering of events that do not involve the processor. This observation is interesting because some architectures that implement a relaxed memory model (such as the Power PC) need expensive global barrier operations to obtain SC. Such barriers are not required for CCCC.

We believe this argument has merit. However, it obliges programmers to reason about *all* programs (programs with races or without races) using CCCC rather than SC. We look forward to research establishing reasoning principles for CCCC, and investigating the productivity and performance tradeoffs between SC and CCCC. We point out that the RAO framework presented in this paper can be extended to account for CCCC (in essence by replacing the underlying "happens before" order with a more local binary relation that is not necessarily transitive). It should be possible to develop an appropriate version of the Fundamental Theorem for CCCC models.

No Thin Air Reads? We consider now programs **with** races. Such programs may be deliberately designed – indeed there are a class of concurrent algorithms (e.g.[11]) designed to contain races, in order to enable desirable features such as increased throughput and non-blocking progress. Alteratively, programs may contain races due to errors. In both cases it is necessary that the semantics of programs with races be precisely defined – in the former so that programs can be shown correct, and in the latter so that programmers have some chance of debugging the problem.

For such programs different answers are possible. Consider a language such as C++ in which programs with races are considered to be erroneous and their behavior is undefined. In such a case all transformations should be permitted as long as only programs with races can distinguish between them. For instance, it should be possible to replace any write `x=y` with the i/o equivalent `x=42;x=y`. Only a program with a race would be able to see the "out of thin air" write `42`.

Such a transformation is not as unreasonable as it may appear. For instance a vectorizing compiler may wish to pack multiple variables `x,y,z,u` into two long words and use vector instructions to optimize execution. The code `x=1;y=1;z=1;u=0` may be implemented with the code sequence `x,y,z,u=1,1,1,1;u=0`. Now the implementation has introduced a Thin Air Write `u=1` which can be detected by a program with races.

On the other extreme are languages such as Java which satisfy the property that certain data types, such as object references, behave like capabilities. A piece of code can obtain a reference r to an object only if it creates the object or it reads a memory location containing that reference. The integrity of large applications written in such languages relies on the property that references to objects can be "closely held", i.e. held only by a certain collection of programmer-specified objects. A semantics which permits Thin Air Reads would permit an attacker to introduce code into the system (e.g. with an applet) which may gain access to such a closely held object via some sequence of seemingly innocuous transformations.

A litmus test for "No Thin Air Reads" is the following test case from [12]. (For the convenience of the reader we indicate with each example the corresponding test number in [12] using the (TC xx) notation. For now, we use an informal notation for programs. We formalize the syntax in Section 2.)

EXAMPLE 1 (TC 4) See also [10, Fig 2]. Consider the program

```
x=0;y=0;(r1=x;y=r1|r2=y;x=r2)
```

("`;`" binds more tightly than "`|`".) Such a program may not exhibit the behavior `r1==r2==1`; values are not allowed to materialize out of thin air. □

A related case is exemplified by the following variant of TC 2.

EXAMPLE 2 Consider the program

```
x=0;y=0;(r1=x;r2=x;y=(r1==r2)?1;|r3=y;x=r3)
```

[1] Two steps are in a race if both read or write the same variable x, at least one of them writes to x and the steps are not ordered by *hb*.

[2] Throughout this paper, when we say "implementation" we mean the compiler/run-time system/architecture/hardware—that is, all elements of the language implementation.

Such a program should not exhibit the observation `r1==0, r2==r3==1`, since the only justification for `r3=1` appears to require `r1==r2`. □

Note though that it is not difficult for a compiler to transform the program above so that the behavior *is* possible. For instance, it replaces the first thread with the i/o equivalent code sequence `y=1;r1=x;r2=x`. An SC execution yields this result, e.g. via:

```
x=0;y=0;y=1;r1=x;r3=y;x=r3;r2=x;x=1
```

In summary, we shall require that a framework for memory models must be flexible enough to permit the formulation of memory models that answer these test cases differently. Such a framework permits programming language designers to choose a variation appropriate for their language.

Inlinability. Another important requirement arises for the memory model for X10[14, 3] like languages that encourage the use of asynchrony. Any particular implementation is likely to have fewer hardware threads than the number of activities spawned by the computation. Therefore it is necessary for the implementation to ensure that activities are aggregated. Such chunking should not impose any additional runtime cost because of extra synchronization. Therefore we require that the memory model support the ability to "inline" activities, wherever this does not cause deadlock.

Usability. Programmers need to use the memory model to understand all possible behaviors of their programs. Programmers understand programs: hence, as far as possible, a memory model should be presented in terms of a few simple permitted transformations of programs that generate permitted behaviors. A programmer should be able to calculate all possible behaviors of a program by systematically applying these transformations.

Requirements summary. We may now summarize the memory model requirements for X10 like languages. These requirements are based on the fundamental assumption that the responsibility for ensuring that a program is correctly synchronized lies with the programmer. The memory model framework must:

1. Ensure that every model satisfies the Fundamental Property.
2. Be flexible enough to permit different formulations of the "No Thin Air Reads" principle.
3. Permit unrestricted use of single-thread optimizations (e.g. code reordering), subject to the two previous conditions.
4. Require the introduction of explicit memory synchronization operations, such as fences, only as necessary to implement explicit synchronization operations in the language (e.g. `atomic`, `when`, `clocks`).
5. Permit SC-valid program rewrites.

   Given a program P and a single-threaded fragment C, let S be the set of all stores in which C can be executed, considering only SC-executions of P. Let C' be another single-threaded program fragment which produces the same result as C when executed in any store $s \in S$. Then C can be rewritten to C'.
6. Specify a few rules that can be used by the programmer to systematically enumerate all possible execution sequences for a given program (particularly if it has races).

   These rules must take into account potentially JITted implementations of the language; that is, they should permit interleaving of "execution" steps and "compilation" steps (program transformations supported by whole program analysis from the current state in the computation).

Memory models satisfying the above criteria support the following programming methodology:

- Most programmers should use explicit synchronization operations (`atomic`, `when`) or volatile variables to reliably communicate values between activities via shared variables.
- For better performance, programmers may use unsynchronized access to variables provided they ensure the global property that there are no data races involving these variables. They may then reason about their program using sequential consistency.
- If the program contains data races, the programmer/compiler may use the set of rules specified by the memory model to determine permissible execution sequences.

1.2 The basic model

We briefly present the central ideas underlying the memory model, deferring formal details to the main body of the paper.

The central technical idea behind in this paper is to (a) formalize sequential execution through the notion of a *step*, (b) specify a *process* as a partially ordered multiset(=pomset) of steps, together with a binary *linking* operation that specifies how the reads of a step are satisfied, (c) specify "execution" as a binary relation on processes induced by the application of process-to-process *transformations*. A *completed* process is one in which every step that performs a read is linked to a step that supplies the value for that read. The result of executing a program is any completed process that can be obtained from the original by applying zero or more transformations (in any order).

By defining execution through the notion of applying transformations, and designing transformations to reflect both "runtime" actions and compile-time program transformations (which may require whole program analysis), the model presented in this paper accurately reflects the flexibility of Java-like languages with Just-In-Time compilers (JITs).

Steps. Intuitively, a step is a kind of sequential function (hence a "semantic" entity) which reads and writes variables in a store, and performs computations on them. Steps should be closed under sequential composition—given two steps s_1 and s_2 there should be an operation (typically written "$\circ$") which yields a new step that reflects the effect of executing s_1 and then s_2. Each programming language will come equipped with its own notion of primitive, indivisible (atomic) steps (e.g. read or write a 32-bit variable), and with a translation function which maps programs in the language (syntax) to sequences of such primitive steps (semantics).

Let us say that a partial store is an assignment of values to variables; such a store is *total* if it specifies values for all variables. A crucial move is to consider a step to be a *partial write function*. For a sequence of statements s, $pw(s)$ is the partial function from partial stores to partial stores which is defined on an input store d only if d specifies values for all the variables that are read by s, and it maps such a d to the set of *writes* produced by running s. Thus a step carries more information than just the i/o function—intuitively, it records the set of variables read as well as the set of variables actually written by the program. For instance the behavior of the program `skip; x=x` is different from `skip` (even though both have the same i/o function), since the former can cause a race whereas the latter can not. Similarly the program `x=y;x=z` is different from the program `x=z` (even though both have the same i/o function) since the former may be involved in a race with `y` but the latter can not. Partial write functions make these extra distinctions while being able to recover the i/o function, if needed.

With such a view of sequential execution in hand, the notion of concurrent execution is easy to define: it is a multiset of steps with two bits of additional structure. First there must be a partial order on steps arising from sequentiality of steps executed by the same thread (the "happens before" order). Second, there must be a way to reflect *links* that record which step f was used to answer

the read of a variable x by a step g. The links must satisfy a consistency condition with the happens before relation, namely, if a link connects a step f to a step g on a variable x, then either f and g are unordered or there is no other step between f and g (in the *hb*-order) which writes on x. (This condition is called the hb-*consistency condition*.)

Transformations. A *process* is taken to be a set of partially ordered sets (=pomsets) with links, closed under a certain set of simple transformations.

All models in the RAO (Relaxed Atomic + Ordering) family are equipped with the transformations *IMprovement* (IM), *COmposition* (CO), *LInk*(LI), *PRopagation* (PR), and *AUgmentation* (AU). Additionally, each model has a *DEcomposition transformation* which we will generically call DX. DX is required to refine the "weakest" decomposition transformation, DL.

The central transformation is LI, a "run-time" action that permits a step to "read" the information in another *hb*-unordered steps by introducing a *link*. Let f and g be two *hb*-unordered steps. LI is parameterized by a non-empty set of variables W. It links f and g so that information produced by f on W is used to update the input store into g. Thus g "sees" the writes on W performed by f—even though f is not *hb*-ordered before f. Thus LI is able to take advantage of a write to a variable x that is in a race with a read of x. We shall see in Section 4.1 that the use of LI distinguishes raw variables from volatile variables; volatiles do not permit the use of LI.

CO permits two successive steps $g;h$ to be replaced by $f = g \circ h$, as long as incoming and outgoing links and *hb* edges are respected. CO may be thought of as a compile-time step that reflects the compiler's decision to execute two steps together as a single step.

Conversely, DL permits the compiler to break up a step f into a pair of atomic steps g followed by h as long as $f = h \circ g$[3] and incoming and outgoing links are respected. While this restriction is strong enough to guarantee that no new races are introduced, it permits the replacement of `x=y` by `x=42;x=y` and hence invalidates Examples 1 and 2. Other decomposition rules are permitted, however they must all strengthen DL (i.e. impose extra conditions).

The decomposition rule for a programming language specifies the intermediate reads and writes that can be performed when decomposing a composite step. The requirement that decomposition rules strengthen DL ensures that they cannot introduce new races. Nevertheless, for programs *with* races, DL offers opportunities for a step to observe values produced internally during the execution of another step. Therefore the programming language designer must carefully choose a decomposition rule appropriate for the language.

AU is the only transformation that changes the *hb* relation between existing steps. An *hb*-edge can be added between two steps provided that the result of the transformation is a valid process. This transformation reflects a compiler's decision to schedule two unordered step one after the other. This may now introduce more opportunities to answer reads through CO. This transformation is not supported by the Java Memory Model described in [10].

PR is a generic "whole program" transformation. It permits a step f to be replaced by a step g provided that f and g are equivalent in all stores that satisfy a condition c, and it is the case that all SC executions of the program force the condition c to be true before f is executed. PR permits whole program analysis to be factored into the model.

IM permits a step f to be replaced by a step g if $io(f) = io(g)$, and g reads and writes fewer variables than f, while respecting all incoming and outgoing links. IM permits extra reads and writes to be dropped (e.g. `x,y=x,2` to be replaced by `y=2`). It may be thought of as a compile-time step that permits a compiler to replace a code fragment by a "better" code fragment, getting rid of unnecessary operations.

Additional remarks. CO, DX, IM, AU and PR are compatible with a totally-ordered notion of memory—memory is a global set of locations from which every read fetches the current value and every write modifies the current value. However, LI permits the same thread to see two different writes if it performs two different reads of the same variable in sequence, even if no other thread has taken any action in the meantime. Thus, LI is *not* compatible with such a "Totally Ordered Memory" principle [1]. Indeed, LI does not even support the notion of "a central store" shared by all threads (and therefore does not support "coherence"). Two threads may simultaneously read two different values for the same variable.

Motivations for these transformations. Informally, let us say that a transformation X is *well-behaved* if for all well-behaved processes P (processes whose SC executions have no races), $P(X)$ is well-behaved and the set of SC executions of $P(X)$ is contained in those of X. We shall see in Section 2.5 that all six transformations are well-behaved; this is the basis for the Fundamental Theorem. Let us say that a transformation is *SC* if for any process X, the SC behaviors of $P(X)$ are contained in those of X. The transformations CO,AU,IM and PR are SC. LI and DX are not—they are the core "weak" (non-SC) transformations around which RAO is built.

There is is nothing particularly singular about the choice of CO, AU, PR and IM, other than they appear to be a fairly simple, small, orthogonal basis for SC transformations performed by compilers and provide a concrete way (e.g. exhaustive enumeration) to establish that a certain behavior is *not* exhibited by a given program. We leave as future work a more general axiomatic formulation of the RAO models which will permit any SC transformation to be admitted, as long as it satisfies certain conditions.

Volatile. On this basis various synchronization operations can be defined. Volatile variables introduce synchronization conditions. In this paper, we consider three variants of volatility, one introduced in JLS 2 (the weakest), DX-restricted volatility, and JLS 3 volatility.

1.3 Related work

Location consistency model. Location consistency (LC) [4] is probably the weakest memory model described in the literature. The distinguishing property of LC is that it does not rely on coherence, thus dispensing the need for cache snooping and directories in a multiprocessor implementation. Gao and Sarkar argue that the model is equivalent to release consistency (RC) [5] for programs that are data race free. However, unlike RAO the specification of LC is not suited as a basis for a memory model of a high-level programming language as it does not explicitly define which re-orderings of access and synchronization statements are permitted [18]. Like LC, RAO does not rely on the coherence assumption.

OpenMP and UPC memory models. The memory models of OpenMP [7] and UPC [19] have been specified after the original specification of these language extensions. The fundamental difference with RAO is as follows: Both OpenMP and UPC are founded on programming languages with unsafe typing and pointer arithmetic and thus the requirements that their memory models impose on programs that are *not* data race free can be looser. RAO, in contrast, is designed for type safe-languages like X10 or Java with the strong memory safety in mind. The focus of the specification of the UPC and OpenMP memory model is on the effect and ordering guarantees provided by certain accesses with synchronization semantics and explicit synchronization constructs—not on guarantees that are given in the absences of such synchronization. Both models allow the introduction of spurious writes, and reads may observe "out of thin air" values in programs with data races [2].

[3] Note: We define $\circ$ to use application order, $f \circ g = \lambda d.(f(g(x)))$, rather than textual order, $f \circ g = \lambda d.(g(f(x)))$.

Java memory model. The RAO model can be thought of as a "happens before" model, discussed in [10, Section3]. RAO is generative, given a source program it generates all possible sequences of executions. In contrast, the methodological stance of [10] is that a trace must be given beforehand; the memory model is then specified in terms of which traces are correct. We feel that valuable information is lost when one moves from a generative model to an oracle; in particular, the task of specifying the semantics is made harder.

ASIAN 2004 paper. This paper generalizes and simplifies [13]. The core concept of linking is derived from the action sets of [13]. The "unique valuation" condition has been replaced by the simpler well-foundedness condition. Conditional linkings have been done away with in favor of (partial) steps. The formulation of the model in terms of a set of permitted transformations is new to this paper.

Store Atomicity Arvind and Maessen [1] have recently proposed a framework for the definition of memory models. Their work shares conceptual commonalities with this work: both are operational and allow enumeration of possible program behaviors based on transformation rules. However, [1] is focused on processor memory models that give semantics to multiple instruction streams with load and store access. Since processors perform a fairly limited set of "transformations" on instruction streams (basically reordering), the baseline of the [1] model can be relatively strong (i.e., store atomicity, which requires that all accesses are serializable). On the other hand, RAO is focused on providing a precise account for memory models for *programming languages*, and must hence account for a wide variety of program transformations and their interactions with the memory model.

1.4 Rest of this paper

The next section presents the basic formal definitions and results of the paper for the unsynchronized language. Section 3 presents several examples to illustrate the range of applicability of the model. All these examples—and all the examples in [12]—can be treated formally within our model. Section 4 treats various definitions of volatiles.

In this extended abstract, all proofs are elided. We refer the reader to the full version of the paper for a complete formal development.

2. RAO Model

2.1 Preliminaries

First some simple preliminaries to fix intuitions.

Syntax. To make the following discussion concrete, we now introduce a syntax for steps of single-threaded code. The syntax is intended to be illustrative, to have a core set of constructs into which a concrete programming language can be translated so that its memory model can be defined. It may be extended routinely with concepts such as function definitions.

(Variables)	x	::=	$\mathtt{x} \mid \ldots$
(Condition)	c	::=	$\mathtt{true} \mid \mathtt{false} \mid e{=}{=}e \mid c\&\&c \mid \mathtt{!}c$
(Expression)	e	::=	$\mathtt{k} \mid x \mid c\,\mathtt{?}\,e\,\mathtt{:}\,e \mid c\,\mathtt{?}\,e \mid (e)$
(Step)	s	::=	$\bar{x}{=}\bar{e}$

The language is simple. It permits partial definition of terms—with one-sided conditional branches (`c?e`) and two-sided conditional branches `c?e1:e2`. The term `c?e` is undefined if `c` evaluates to `false`.

We will write `skip` for the step $\varepsilon = \varepsilon$.

Stores. By a *partial function* from a domain D to a range R we shall mean a function that is defined from some subset of D, $dom(f)$, into R. The *restriction* of a partial function f to a set V, $f \downarrow V$, is f restricted to the domain $dom(s) \cap V$.

We fix an infinite set of variables V and a set of values L. A *partial store* d is a partial map from V to L, a *total* store is one whose domain is V. We designate the set of all partial stores by *Store*, and the set of all total stores by *TStore*. We treat a store isomorphically as a set of bindings, $\{x_0 = v_0, x_1 = v_1, \ldots\}$.

The *union* $d_0 \cup d_1$ of two stores d_0 and d_1 (with disjoint domains) is their union when viewed as a set of bindings. Since two stores may have conflicting information, their *asymmetric union* $c[d]$ (read as: *c updated by d*) is quite important and is defined as the set of bindings in d together with the bindings from c for those variables not bound by d.

We define a binary relation on stores $c \leq d$ (read as: *d extends c*) to hold iff $d[c] = d$. It is easy to see that $\leq$ is a partial order. Note that for distinct stores d, d', $d \leq d'$ implies that $dom(d)$ is strictly contained in $dom(d')$.

Functions on stores. Define a partial order on partial functions over stores by: $f \leq g$ if $dom(f) \subseteq dom(g)$ and $f(c) \leq g(c)$ for all $c \in dom(f)$. The notion of *monotonicity* of such functions is standard. f is *monotone* if $d \in dom(f), e \geq d$ implies $e \in dom(f)$ and $f(e) \geq f(d)$. For any function f on stores, we define its *transition function* $f^\sharp$ by: $f^\sharp(c) = c[f(c)]$. Unlike f, $f^\sharp$ "flows" the input through to the output.

A function f is *complete* if it is defined for every total store.

2.2 Modelling single-threaded code

The fundamental intuition underlying the models is that a piece of sequential code should be modelled as a *step*, i.e. a *function* from stores to stores. We use **partial functions** that record for each **partial store** d the writes produced by executing s on d. In the rest of this discussion, given a syntactic step s, we will use $[\![s]\!]$ for its denotation, i.e. the function associated with the step. On an input store d, the output store $[\![s]\!](d)$ is defined at variable x only if x is written by s. Dually, if $[\![s]\!](d)$ is not defined on a given (partial) store d, then it must mean that s must read some variable that does not have a binding in d.

EXAMPLE 3 $[\![\mathtt{x} = (\mathtt{false?0})]\!]$ is the unique function that is defined on every input (i.e. it does not perform a read) and maps it to $\{\}$. In no store can the assignment to `x` happen since its precondition, `false`, can never be satisfied.

$[\![\mathtt{x} = 1]\!]$ is the function that is defined on every input and maps it to $\{\mathtt{x} = 1\}$.

On any input store d, $f = [\![\mathtt{x} = (\mathtt{y} == 1\mathtt{?}1)]\!]$ is a function that must definitely read `y`, hence d must define a value for `y`. The function produces a write on `x`, `x=1`, iff $d(\mathtt{y}) = 1$. Formally, $d \in dom(f)$ iff $\mathtt{y} \in dom(d)$. f maps such a d to $\{\mathtt{x} = 1\}$ if $d(\mathtt{y}) = 1$ and to $\{\}$ otherwise.

$[\![\mathtt{x} = (\mathtt{y} == 1\mathtt{?}0 : (\mathtt{y} == 0\mathtt{?}1))]\!]$ is defined on input stores d iff $\mathtt{y} \in dom(d)$. Such a d is mapped to $\{\mathtt{x} = 0\}$ if $d(\mathtt{y}) = 1$, to $\{\mathtt{x} = 1\}$ if $d(\mathtt{y}) = 0$ and to $\{\}$ otherwise.

$f = [\![x, r = (x! = 42\mathtt{?}42), 42]\!]$ is defined on input stores d iff $\mathtt{x} \in dom(d)$. Such a d is mapped to $\{\mathtt{x} = 42, \mathtt{r} = 42\}$ if $d(\mathtt{x})! = 42$ and to $\{r = 42\}$ if $d(\mathtt{x}) = 42$. Note that for all $d \in dom(f)$ we have $\{\mathtt{x} = 42\} \leq f^\sharp(d)$ – in some cases because of the write in $f(d)$ and in some cases because of the flow-through from the input. Our treatment of steps as partial functions enables us to model this distinction.

□

We further restrict our attention to sequential functions that correspond to the execution of single-threaded code. Such code must

perform its basic operations (e.g. reads and writes) in sequence, one after the other. (It may not perform operations such as a "parallel or", which reads reads two variables in parallel, without specifying the order.) Therefore such sequential functions f have the property that any store d is either in f's domain, or there is a non-empty set of variables, $n(f,d)$, all variables in which must be read *next* by the function. The formal definition of $n(f,d)$ is standard [17] and is elided in this extended abstract.

DEFINITION 1 (STEP). A *step* is a monotone, sequential, partial function from finite stores to finite stores.

2.3 Modeling concurrent programs

A concurrent program can now be thought of as a *partially ordered multiset (pomset)* of steps. The partial order is called the *happens before* order and indicates those steps that are known to occur before other steps. Formally, an AO process is a initialized pomset of steps, with a possibly empty set of links:

DEFINITION 2 (SEQUENTIAL COMPOSITION). Given two steps f and g, their *sequential composition* $g \circ f$ is the partial function defined only on those d s.t. $d \in dom(f)$ and $f^\sharp(d) \in dom(g)$ and which maps d to $f(d)[g(f^\sharp(d))]$.

While the definition of a step captures only the actual output of the step, the use of a step in a sequential composition permits inputs to traverse untouched to the output of the first step, if they are needed by the second step. However, the output produced by the composite is only the (combination of) output produced by each step—flow through from the input is not counted as output. It is not hard to see that $(f \circ g)^\sharp = f^\sharp \circ g^\sharp$. We now consider examples of sequential compositions of steps.

EXAMPLE 4 Consider $f = [\![\mathtt{x} = 1]\!]; [\![\mathtt{y} = 1]\!]$. It is not difficult to see that $f = [\![\mathtt{x},\mathtt{y} = 1,1]\!]$. Formally, one uses the definition of denotation of a step given above, and the definition of composition of steps (Definition 2) to establish this.

Let us consider a step that reads a variable after conditionally writing into it: $f = [\![\mathtt{x} = (\mathtt{x} == 1?0 : 1)]\!]; [\![\mathtt{y} = \mathtt{x}]\!]$. Clearly this should be the same function as $[\![\mathtt{x},\mathtt{y} = (\mathtt{x} == 1?0 : 1), (\mathtt{x} == 1?0 : 1)]\!]$. Again, this can be established formally.

In general, $[\![\mathtt{x} = \mathtt{c}?\mathtt{z}]\!]; [\![\mathtt{y} = \mathtt{x}]\!]$ is the same as $[\![\mathtt{x},\mathtt{y} = \mathtt{c}?\mathtt{z}, \mathtt{c}?\mathtt{z} : \mathtt{x}]\!]$. Specifically $[\![\mathtt{x} = (\mathtt{x}! = \mathtt{k}?\mathtt{k}); \mathtt{y} = \mathtt{x}]\!]$ is the same as $[\![\mathtt{x},\mathtt{y} = (\mathtt{x}! = \mathtt{k}?\mathtt{k}), \mathtt{x}! = \mathtt{k}?\mathtt{k} : \mathtt{x}]\!]$, and $[\![\mathtt{x},\mathtt{y} = (\mathtt{x}! = \mathtt{k}?\mathtt{k}), \mathtt{k}]\!]$. □

A rule for calculating the sequential composition of steps is formalized in the full version of the paper.

DEFINITION 3 (LINK). Given a pomset of steps P, a *link* is a quadruple (s,t,x,v) where $s,t \in P$, x is a variable and v is a value.

DEFINITION 4 (INPUT STORE, LINK-COMPLETED STEP). Given a set L of links $(\_,s,x,v)$ entering s, $in(s)$ is the store $\{x = v \mid (\_,s,x,v) \in L\}$. When used as a function, $in(s)$ stands for the function that maps input d to $d[\{x = v \mid (\_,s,x,v) \in L\}]$.

We say that a step s is *complete* if $in(s) \in dom(s)$.

We define $s^\dagger$ (read: *link-completed* s) as the function $s \circ in(s)$.

EXAMPLE 5 Consider a link $(f,g,x,1)$.
Let f be $[\![x = 1]\!]$ and g be $[\![r = x]\!]$. Then $g^\dagger$ is $[\![r = 1]\!]$. f can be used to answer the read on $\mathtt{x}$ in g, but f's outputs are not propagated.
Let f be $[\![x,y = 1,1]\!]$ and g be $[\![r = x]\!]$. Then $g^\dagger$ is $[\![r = 1]\!]$. Irrelevant information in f is ignored.
Let f be $[\![x = 1]\!]$ and g be $[\![r,x = x,(x! = 1)?1]\!]$. Then $g^\dagger$ is $[\![r = 1]\!]$. Information in f may force a write of g to be dropped. □

DEFINITION 5 (WRITE-BEFORE). Let P be a pomset of steps. For steps $f,g \in P$ and a variable x, define $f\ wb_x\ g$ (read: g can read x from f) if (i) f writes x, i.e. $x \in dom(f(\{\}))$, and (ii) f and g are unordered, or $f\ hb\ g$ and (iii) there is no other step f' between f and g (in the hb-order) s.t. $x \in dom(f'(d))$ for any store d.

DEFINITION 6 (AO PROCESS). An AO process (P,Ls) is a partially ordered multiset of steps P, together with a set Ls of links satisfying:

Link Uniqueness $(s,t,x,v),(s',t,x,v') \in Ls$ implies $s = s'$ and $t = t'$.

Link Well-definedness $(s,t,x,v) \in Ls$ implies s is complete and $s^\dagger(in(s))(x) = v$. (Thus s unconditionally produces v for x, given its input links.)

Link Acyclicity The graph with steps as nodes and edges $s \to t$ if $(s,t,\_,\_) \in Ls$ is acyclic.
Note that this condition is not the same as hb-acyclicity. Indeed, an edge may be introduced by a link when the two steps are unordered by hb.

HB Consistency $(s,t,x,v) \in Ls$ implies $s\ wb_x\ t$.

Initialization Condition: If a step in P touches a variable $\mathtt{x} \in V$ then there is a unique step in P that writes into $\mathtt{x}$, does not read from $\mathtt{x}$, and hb any other step in P that touches $\mathtt{x}$.

It is useful to visualize an AO process as a directed graph with nodes labeled with steps and edges representing the hb relation.

DEFINITION 7 (COMPLETED AO PROCESS). An AO process A is said to be a *completed execution* if every step of A is complete.

DEFINITION 8 (SC (EXECUTION OF) AO PROCESS). An AO process A is said to be *sequentially consistent* (SC) if its hb order is total. An *SC execution* of an AO process A is any SC AO process A' with the same set of steps and link-set as A. For an SC process P with steps $s_0,\ldots,s_{n-1}$ enumerated in hb order, the i/o function of P, $io(P)$ is $io(s_0^\dagger \circ \ldots \circ s_{n-1}^\dagger)$.

DEFINITION 9 (WELL-BEHAVED AO PROCESS). An AO process P is *well-behaved* if no SC execution Q of P has a P-race. Q has a P-race if in the execution $s_0^\dagger \circ \ldots \circ s_{n-1}^\dagger$ of Q there are steps s_i and s_j $(i < j)$, such that s_i produces a value that s_j reads, but it is not the case that $s_i\ hb\ s_j$ in P.

Process combinators

AO processes are composed using "`;`" (sequential composition) and "`|`" (parallel composition). `;` binds more tightly than `|`.

P ; Q has the steps of P and Q with the hb order of P and Q extended to ensure that every step of P hb every step of Q.

$P \mid Q$ has the steps of P and Q with the hb order of P and Q.

Note that `;` is associative, whereas `|` is commutative and associative (but not idempotent—the resulting pomset has twice as many steps). If we use `skip` to denote the unique process with no steps, then $\mathtt{skip} \mid P = P \mid \mathtt{skip} = P$, and $\mathtt{skip};P = P;\mathtt{skip} = P$.

2.4 Transformations of AO processes

In the RAO model, the following transformation rules can be used to transform an AO process. The transformation is applicable only if the resulting structure is an AO process.

The transformations IM, AU, CO, DL and LI are local, i.e. the applicability of the transformation does not depend on whole program analysis or on the absence or presence of other steps than the ones named in the transformation.

Below, for a process (P,Ls) with steps $p \in P, p'$ when we say *replace p by p' while preserving all edges and links* we mean that a

new process (P', Ls') is created in which P' is the same as P with p replaced by p', every edge $(h,p) \in hb$ is replaced by (h,p'), every edge $(p,h) \in hb$ is replaced by (p,h'), every link $(q,p,x,v) \in Ls$ is replaced by (q,p',x,v), and every link $(p,q,x,v) \in Ls$ is replaced by (p',q,x,v).

2.4.1 Improvement

We say that *a step g improves a step f* if $io(g) = io(f)$, $dom(g) \subseteq dom(f)$, and $f \geq g$. The first condition ensures that the behavior of f and g under sequential (sequentially consistent) execution is identical. The second condition ensures that *extra reads*—reads of variables that do not affect the final result—can be dropped. The third condition ensures that *extra writes*—writes of the form `x=x` – can be dropped. Let us write $[\![s]\!]$ for the step corresponding to a piece of sequential code s. Then $[\![\mathtt{x} = \mathtt{y}]\!]$ improves $[\![\mathtt{x} = \mathtt{z}; \mathtt{x} = \mathtt{y}]\!]$ and $[\![\mathtt{x} = \mathtt{y}; \mathtt{z} = \mathtt{z}]\!]$.

DEFINITION 10 (IM). Given an AO process (P,Ls), replace $f \in P$ with a step g while preserving all edges and links, if g improves f, and g writes on every variable x for which $(f,\_,x,\_) \in Ls$.

2.4.2 Augmentation

DEFINITION 11 (AU). Add an *hb*-edge between two steps in P provided that the resulting set is an AO-process.

AU permits the implementation to schedule two otherwise unconstrained steps (belonging to separate threads) in a particular order.

2.4.3 Composition

Consider two steps $f;g$. We would like to replace them with $e = g \circ f$ and move the incoming and outgoing links of f and g to e. That is, we would like to replace $h' = g^\dagger \circ f^\dagger$ by $h = (g \circ f)^\dagger$.

The following conditions are sufficient. If f and g have incoming links for x, those links must arise from the same step (so they read the same value and have the same *hb* relationship with the link source). This implies $in(f)[in(g)] = in(g)[in(f)]$, or (in terms of functions) $in(f) \circ in(g) = in(g) \circ in(f)$. Further, f should pass through, without modification, any variable for which there is a link into g. Symmetrically, if f has an outgoing link for x, then g should pass through the value produced by f on x without modification. This motivates the following definition.

DEFINITION 12 (CO). Let (P,Ls) be an AO process. Let the immediate *hb* successor of f in P be g (and only g), and the immediate *hb* predecessor of g be f (and only f). Let $h = g \circ f \circ in(g) \circ in(f)$ and $h' = g \circ in(g) \circ f \circ in(f)$. Let f and g satisfy the property that (i) $(s,f,x,v),(s',g,x,v') \in Ls$ implies $s = s'$ (and therefore $v = v'$), (ii) $h = h'$, (iii) for every x s.t. $(f,\_,x,v) \in Ls$, $h(in(f)[in(g)])(x) = f(in(f)[in(g)])(x)$.

Replace f and g by $e = g \circ f$, replacing each link/edge entering/exiting f or g by the same link/edge entering/exiting e.

CO permits the implementation to schedule two successive steps in the *hb*-order together, treating them as part of the same sequential step. In the new process $e^\dagger$ is the same function as $g^\dagger \circ f^\dagger$ in the old process. Further, the conditions are always satisfied if g has no incoming links and f has no outgoing links.

2.4.4 Propagation

By a constraint q on stores we mean a (possibly infinite) set of stores. A store d satisfies q if $d \in q$. Two functions f_0 and f_1 on stores are q-equivalent if for $Q = q \cap dom(f_0)$, we have $Q = q \cap dom(f_1)$, and $f_0 \downarrow Q = f_1 \downarrow Q$.

DEFINITION 13 (PR). Let $A = (P,Ls)$ be a process. Let $f \in P$ and f' be a step that is q-equivalent to f, where in all SC-executions (Definition 8) of P, q is true at (before) f.

Replace f by f', preserving all edges and links.

PR permits an implementation to perform any global optimization based on data-flow analyses as long as the analyses consider only SC executions. Since this transformation effects a global analysis, it is sensitive to the presence steps in P other than s. One of its uses is to replace conditional execution with unconditional execution.

We shall see below that typically an application of CO enables applications of DL. Applications of PR and AU enable applications of CO. Applications of DL enable applications of AU, etc.

2.4.5 Link

LI is an "inter thread" version of CO.

DEFINITION 14 (LI). Let $A = (P,Ls)$ be a process. Let $s,t \in P$ and x be a variable s.t. (i) $s\ wb_x\ t$ in P, (ii) s is completed, (iii) $s(in(s))(x) = v$, and (iv) $x \in n(t,in(t))$.

Transform A to $(P, Ls \cup \{(s,t,x,v)\})$.

EXAMPLE 6 (see also Example 5) Let f be `x=(x!=42)?42` and g `r=x` be two steps in an AO process (P,Ls) and the only link entering g is from f and labeled with x. Then `r=42` is an improvement of $g^\dagger$ (since it does not read `x`). Thus, a conditional write in f may result in an unconditional write by g. □

2.4.6 Decomposition

DEFINITION 15 (DL). Let (P,Ls) be an AO process, $f \in P$ s.t. $f = h \circ g$, and for every incoming (outgoing) x-link for f it is the case that precisely one of f or g reads (writes) x. (Call that step i_x.)

Replace f with $g;h$. Every edge $(e,f) \in hb$ is replaced by (e,g) and every edge $(f,e) \in hb$ by (h,e). Every link (e,f,x,v) and (f,e,x,v) in Ls is replaced by (e,i_x,x,v) and (i_x,e,x,v) respectively.

Intuitively, the implementation decides to break up a single step f into two steps g and h since the behavior of a thread executing f is indistinguishable (in any race-free context) from the behavior of the thread executing first g and then h.

DR adds to DL the condition that for any variable x and input store d, x is in the domain of at most one of the stores $g^\dagger(d)$ and $(g^\dagger \circ h^\dagger)(d)$. DW adds to DL the condition that for any variable x and input store d, x is in at most one of $n(g^\dagger,d)$ and $n(g^\dagger \circ h^\dagger,d)$. DO adds both these conditions to DL. We let DX stand for any of these four decomposition rules.

In combination with CO, DX may change the *hb* order of the original program. For instance consider the program fragment `x=1;y=2` . Using CO this may be converted to `x,y=1,2` and then using DX to `y=2;x=1`. Thus the original *hb* order is inverted. Some synchronization constructs (e.g. volatiles) are designed to ensure that such reordering cannot occur (see Section 4.2); hence their semantics places restrictions on the application of DX.

DX is also useful in conjunction with LI: sometimes it is possible to break a function f which performs some reads into f_0 and f_1 in such a way that f_0 does not perform any reads. Now f_0 can be used as a source for a link.

2.4.7 RAO process

DEFINITION 16 (RAO PROCESS). An RAO *process* is a set of AO processes closed under CO, DL, IM, LI, PR and AU. For any AO process P, the smallest (qua set) RAO process containing P is denoted by RAO(P).

2.5 Main theorem

Let P, Q be AO processes. Say that $P \xrightarrow{X} Q$ if Q is obtained from P by the application of a transform X in the set of RAO transformations. The *SC i/o functions* of P, $sc(P)$ is the set of functions $io(Q)$ for all SC executions Q of P (Definition 8).

Let $clo(P)$ represent the set of AO processes obtained from P by zero or more applications of the given transformation. We take the *observations* of a process P to be the set of *i/o functions* of P, $io(P)$ defined as the set $\{f \mid f \in sc(Q), Q \in clo(P), Q \text{complete}\}$.[4] We say that $O \in io(P)$ has *a proof of size n* if there is an $\xrightarrow{X}$ sequence of length n from P to a completed process Q such that $O \in sc(Q)$.

LEMMA 17. *For all AO processes P,Q if P is well-behaved and* $P \xrightarrow{X} Q$ *then:*

Good behavior is $\xrightarrow{X}$-invariant. *Q is well-behaved.*

SC behavior is $\xrightarrow{X}$-invariant. $\text{sc}(Q) \subseteq \text{sc}(P)$.

IO behavior is $\xrightarrow{X}$-invariant. $\text{io}(Q) \subseteq \text{sc}(P)$.

THEOREM 18 (FUNDAMENTAL PROPERTY). *Let P be a well-behaved AO process. Then* $\text{io}(P) \subseteq \text{sc}(P)$.

3. Examples

We consider some examples. Note: In analyzing the test cases below we shall usually omit the initial step in the AO process. Further, we shall not be combining (through CO) two steps both of which have incoming links. In such cases it is possible to replace t with $t^\dagger$ whenever a new link (s,t,x,v) is added to the link-set.

3.1 Single-thread reordering

EXAMPLE 7 (TC 7) We illustrate the use of CO, DR and AU to obtain single-thread reordering. Consider the program:

```
x,y,z=0,0,0;(r1=z;r2=x;y=r2|r3=y;z=r3;x=1)
```

Is behavior `r1==r2==r3==1` exhibited? Single-thread optimization could permit `r1=z` to be moved to the end of the thread, and `x=1` to the beginning of the thread. The result would then follow by an SC execution.

Formally this can be analyzed as follows. We show a chain of AO processes each obtained from the previous by applying the noted transformation. The last process exhibits the desired behavior.

Consider the steps `r1=z;r2=x;y=r2`. These may be collapsed into a single step using CO to yield `r1,r2,y=z,x,x`. But this step can be decomposed into `r2,y=x,x;r1=z`—this is the code motion discussed above. Similarly `r3=y;z=r3;x=1` yields through CO and DR `x=1;r3,z=y,y`. Now we can interleave the steps in the appropriate order using AU to accomplish the desired result.

□

EXAMPLE 8 (TC 2) See also Fig 5 in [10]. This example illustrates that CO, DL and AU can simulate the effect of redundant read elimination. Consider the program:

```
x,y=0,0;(r1=x;r2=x;y=(r1==r2)?1|r3=y;x=r3)
```

This should exhibit `r1==r2==r3==1` since redundant read elimination could result in simplifying `r1==r2` to `true`. Subsequently `y=1` could be moved early.

This reasoning is readily formalized as follows. In each step we specify only the links added at that step. By convention the links associated with a step are the union of all the links associated with previous steps, together with the links added at that step.

[4] The two definitions of *io* for SC processes coincide.

```
x,y=0,0;(r1,r2=x,x;y=(r1==r2)?1|r3=y;x=r3)  (CO)
x,y=0,0;(r1,r2,y=x,x,1|r3=y;x=r3)           (CO)
x,y=0,0;(s0:r1,r2,y=x,x,1|s1:r3=1;x=r3)
                                    (LI,(s0,s1))
x,y=0,0;(r1,r2,y=x,x,1|r3,x=1,1)            (CO)
```

□

This example shows that the RAO model permits two reads to be answered by the same write **without determining what that write is**. This is just a consequence of CO—by composing all the steps of Thread 1, we ensure that the reads into `r1` and `r2` will be answered from the input store (for the composite step). Hence they must have the same value. Thus the steps of the first thread are equivalent (as functions) to the single step `r1,r2,y=x,x,1`.

EXAMPLE 9 (TC 3) This example illustrates that the application of CO, DR and AU is not affected by the presence of additional threads. Consider the program:

```
x,y=0,0;(r1=x;r2=x;y=(r1==r2)?1|r3=y;x=r3;|x=2
```

The behavior `r1==r2==r3==1` can be exhibited, using the same reasoning as in Test 8. The additional thread does not interfere with the application of CO and LI. □

EXAMPLE 10 (TC 17) Consider the AO process:

```
x,y=0,0;(r3=x;x=(r3!=42)?42;r1=x;y=r1|r2=y;x=r2)
```

It should be able to exhibit `r1==r2==r3==42` since

$$\texttt{r3} = \texttt{x}; \texttt{x} = (\texttt{r3}! = 42)?42; \texttt{r1} = \texttt{x}$$

and

$$\texttt{r3} = \texttt{x}; \texttt{x} = (\texttt{r3}! = 42)?42; \texttt{r1} = 42$$

have identical i/o functions. But the second program can permit the propagation of `r1=42` to the beginning of the program, resulting in the desired behavior. The RAO analysis mirrors this reasoning:

```
r3=x;x=(r3!=42)?42;r1=x;y=r1|r2=y;x=r2
r3,x=x,(x!=42)?42;r1=x;y=r1|r2=y;x=r2          (CO)
r3,x,r1=x,(x!=42)?42,42;y=r1|r2=y;x=r2         (CO#)
r3,x,r1,y=x,(x!=42)?42,42,42|r2=y;x=r2         (CO)
r3,x=x,(x!=42)?42;s0:r1,y=42,42|s1: r2=y;x=r2 (DL)
r3,x=x,(x!=42)?42;s0:r1,y=42,42|s1:r2=42;x=r2
                                        (LI, s0->s1)
r3,x=x,(x!=42)?42;so:r1,y=42,42|s1:r2,x=42,42 (CO)
s3:r3=42;s0:r1,y=42,42|s1:r2,x=42,42   (LI, s1->s3)
```

(#) In the above example,

$$\texttt{r3}, \texttt{x}, \texttt{r1} = \texttt{x}, \texttt{x}! = 42?42, \texttt{x}! = 42?42 : \texttt{x}$$

and

$$\texttt{r3}, \texttt{x}, \texttt{r1} = \texttt{x}, \texttt{x}! = 42?42, 42$$

denote the same step. In the last line the write to `x` will never be performed by the first step, and hence the write is dropped. □

3.1.1 Inter-thread reasoning—the use of PR

We now consider some examples that illustrate the use of PR.

EXAMPLE 11 (TC 1) This example shows inter-thread reasoning—the use of CO, DE, AU, PR. Consider the RAO process generated from P_0:

$$\texttt{x}, \texttt{y} = 0, 0; (\texttt{r1} = \texttt{x}; \texttt{y} = (\texttt{r1} >= 0)?1 | \texttt{r2} = \texttt{y}; \texttt{x} = \texttt{r2})$$

Arguably, $\text{RAO}(P_0)$ should be able to exhibit `r1==r2==1`. The compiler may determine that `x` and `y` are always non-negative, and hence simplify `r1>=0` to `true`. This allows `y=1` to be moved early. We can formalize this in RAO thus:

```
r1,y=x,r1>=0?1|r2=y;x=r2    (CO)
r1,y=x,1|r2=y;x=r2          (PR#)
```

```
r1=x;s1:y=1|s2:r2=y;x=r2    (DL)
r1=x;s1:y=1|s2:r2=1;x=r2    (LI, s1->s2)
s0:r1=x;s1:y=1|s2:r2,x=1,1 (CO)
s0:r1=1;s1:y=1|s2:r2,x=1,1 (LI, s2->s0)
```

(PR#) Replace `r1,y=x,(x>=0?1)` with the `x>=0`-equivalent step `r1,y=x,1`.

□

EXAMPLE 12 (TC 18) See also [10, Fig 12]. The program:

```
x,y=0,0;(r3=x;x=(r3==0)?1;r1=x;y=r1|r2=y;x=r2)
```

should permit the behavior `r1==r2==r3==1`. A compiler may determine through whole program analysis that the only possible values for `x` are `0` and `1`. Hence if `r3 !=0` it must be the case that `r3==1`. Hence transforming `r1=x` into `r1=1` is legal from the viewpoint of a single thread. But this write can be propagated earlier and SC execution will yield the desired result. The RAO analysis permits this, following the reasoning above.

```
r3,x=x,(x==0)?1;r1=x;y=r1|r2=y;x=r2           (CO)
r3,x,r1=x,(x==0)?1,(x==0)?1:x;y=r1|r2=y;x=r2 (CO)
r3,x,r1=x,(x==0)?1,1; y=r1|r2=y;x=r2
                                    (PR;x in {0,1}
r3,x,r1,y=x,(x==0)?1,1,1|r2=y;x=r2           (CO)
r3,x=x,(x==0)?1;s0: r1,y=1,1|s1: r2=y;x=r2   (DL)
r3,x=x,(x==0)?1;s0: r1,y=1,1|s1: r2=1;x=r2
                                      (LI, s0->s1)
s2: r3,x=x,(x==0)?1;s0: r1,y=1,1|s1: r2,x=1,1(CO)
s2: r3=1;s0: r1,y=1,1|s1: r2,x=1,1   (LI, s1->s2)
```

□

EXAMPLE 13 (Fig 11 of [10]) This test case is *not* permitted by the Java Memory Model described in [10], but is permitted by RAO. Consider the program:

```
x,y=0,0;(r3=x;x=(r3==0)?1|r1=x;y=r1|r2=y;x=r2)
```

Test Case 18 can be obtained from this program by inlining Thread 2 after Thread 1.

```
x,y=0,0;(r3=x;x=(r3==0)?1|r1=x;y=r1|r2=y;x=r2)
x,y=0,0;(r3=x;x=(r3==0)?1;r1=x;y=r1|r2=y;x=r2)(AU)
```

The rest of the derivation follows Case 18. □

3.2 Cross-coupling behaviors

We now consider examples that illustrate *cross-over*.

DEFINITION 19 (CROSS-OVER). Let A be an AO process. A *cross-over* is a set of steps in A that forms a loop in the graph whose nodes are steps and whose edges are links (directed from source to target) or hb-edges.

Naturally, the presence of races, and the use of LI, is critical in establishing a cross-over. The other transformations (CO, DX, IM, AU and PR) are compatible with a totally-ordered notion of memory—memory is a global set of locations from which every read fetches the current value and every write modifies the current value. In our model, such a totally-ordered notion of memory is modelled by the extra condition that n particular, these other transformations are closed on the subset of AO processes satisfying the condition that the linkset is a subset of the happens-before relation. LI does not preserve this additional condition, whereas the other transformations (CO, DX, IM, AU and PR) do.

EXAMPLE 14 (TC 16) See also Fig 1 in [10]. The program:

```
x,y=0,0;(r1=x;x=1|r2=x; x=2)
```

should be able to exhibit the behavior `r1==2; r2==1`. RAO permits it thus:

```
x,y=0,0;(s0:r1=x;s1:x=1|s2:r2=x;s3:x=2)
x,y=0,0;(s0:r1=x;s1:x=1|s2:r2=1;s3:x=2)
                              (LI, s1->s2)
x,y=0,0;(s0:r1=2;s1:x=1|s2:r2=1;s3:x=2)
                              (LI, s3->s0)
```

The final process illustrates the crossover $\{$`s0,s1,s2,s3`$\}$. □

For an example that shows the interleaving of LI and PR is critical, we refer the reader to the full paper.

3.3 No Thin Air Reads behaviors

The following examples involving no thin air reads discuss alternative definitions of the decomposition rule and their consequences. This analysis supports the claim that RAO provides a flexible framework for a programming language designer.

EXAMPLE 15 (TC 4) See also Fig 2 in [10]. Consider the AO process:

```
x,y=0,0;(r1=x;y=r1|r2=y;x=r2)
```

This process should not exhibit `r1==r2==1` even though there is a race. The value `1` cannot be read from thin air.

LI, PR and AU cannot produce the desired result, as can be established by systematically applying them.

Now let us consider various decomposition rules. DO (and hence DL) can establish `r1==r2==1` by:

```
x=0;y=0;(r1=x;y=r1|r2=y;x=r2)
x=0;y=0;(r1=x;y=1;y=r1|r2=y;x=r2)  (DW)
x=0;y=0;(y=1;r1=x;y=r1|r2=y;x=r2)  (DO)
x=0;y=0;y=1;r2=y;x=r2;r1=x;y=r1    (AU*)
y=1;r2=1;r1=1;x=1                    (DO*)
```

However, DR and DO cannot; there is no way of creating the phantom write. □

EXAMPLE 16 (TC 5) Consider the program:

```
x,y,z=0,0,0;(r1=x;y=r1|r2=y;x=r2|z=1|r3=z;x=r3)
```

The behavior `r1==r2==1, r3==0` should be forbidden.

RAO Analysis: As in Test Case 15. The only use of LI will replace `r3=z` with `r3=1`—and this will not give the desired result. An exhaustive case analysis shows that none of the other transformations can produce the desired behavior. □

EXAMPLE 17 (TC 10) Consider the AO program P:

```
x=0;y=0;z=0;
(r1=x;y=(r1==1)?1|r2=y;x=(r2==1)?1
 |z=1|r3=z;x=(r3==1)?1)
```

The behavior `r1==r2==1, r3==0` should not be possible.

This is indeed the case. PR cannot be used to discharge any of the conditionals. CO/DE cannot be used to perform any of the steps of a thread in parallel since there is a read/write dependency. AU can be used to totally order these steps (as would be done in an *sc* execution). But no *sc* execution will give the desired result. LI can be used to replace `r3=z` with `r3=1`, but this will not give the desired result.

□

EXAMPLE 18 (Fig 10, [10]) Consider the program:

```
x=0;y=0;z=0;
(z=1|r1=z;x=(r1==0)?1|r2=x;y=r2|r3=y;x=r3)
```

It should not be possible to observe `r1==r2==r3==1`, since in any "execution" which could exhibit this behavior only Threads 3 and 4 write to `x` and `y`, and hence they cannot manufacture the value `1` out of thin air.

The RAO model validates this reasoning. It is not possible to use PR to reduce `x=(r1==0)?1` to `x=1` (except by using AU to

place `z=1` after the conditional assignment to `x`—but in that case `r1=z` *hb* `z=1` hence `r1` can never see the value `1`). Without that, the only way `r2` can be `1` is for `r1=0` to have been executed before it, but then `r1 !=1`.

LI can be used to transfer `z=1` into `r1=z`; to obtain `r1=1`. However, this will disable the conditional write to `x`. The resulting process cannot produce `1` for `r2` or `r3` since the only writes available produce `0`.

□

EXAMPLE 19 (Example 2 revisited) Consider the program

```
x=0;y=0;(r1=x;r2=x;y=(r1==r2)?1|r3=y;x=r3)
```

Such a program should not exhibit `r1==0,r2==1,r3==1`, since the only justification for `r3=1` appears to require `r1==r2`.

The use of DR (and hence DL) permits `r1==0;r2==1,r3==1`.

```
x=0;y=0;(r1=x;r2=x;y=(r1==r2)?1|r3=y;x=r3)
x=0;y=0;(r1=x;r2=x;y=1|r3=y;x=r3)          (DR)
x=0;y=0;(r1=x;y=1;r2=x|r3=y;x=r3)          (CO*;DO)
x=0;y=0;r1=x;y=1;r3=y;x=r3;r2=x            (AU*)
r1=0;y=1;r3=1;x=1;r2=1                     (CO*;DO)
```

DW also permits the observation:

```
x=0;y=0;(r1=x;r2=x;y=(r1==r2)?1|r3=y;x=r3)
x=0;y=0;(r1=x;r2=x;y=1;y=(r1==r2)?1|r3=y;x=r3)(DW)
x=0;y=0;(r1=x;y=1;r2=x;y=(r1==r2)?1|r3=y;x=r3)
                                           (CO*;DO)
x=0;y=0;r1=x;y=1;r3=y;x=r3;r2=x;y=(r1==r2)?1 (AU*)
r1=0;y=1;r3=1;x=1;r2=1                      (CO*;DO)
```

However, DO alone cannot exhibit this behavior. □

EXAMPLE 20 (Strength reduction) Consider the program:

```
x=1; (r=x;s=x;x=2*r | x=3); u=x
```

Can it yield `u=4`? Here is a derivation:

```
x=1;(r=x;s=x;x=2*r | x=3);u=x
x=1;(r=x;s=x;x=r+r | x=3);u=x  (DO, x=2*r->x=r=r)
x=1;(r=x;s=x;x=r+s | x=3);u=x  (DR)
x=1;r=x;x=3;s=x;x=r+s;u=x      (AU*)
r=1;s=3;x=4;u=4                (CO*;DO*)
```

The use of DR replaces `r=x;s=x;x=r+r` with `r=x;s=x;x=r+s`; DW and hence DO cannot accomplish this. □

4. Synchronization constructs

Synchronization constructs are defined in the RAO model by introducing extra structure to the model, and, if necessary, adding restrictions on the application of various transformations. The basic idea behind synchronization constructs is to introduce mechanisms by which the programmer may reliably communicate values from one thread to another without introducing races, i.e. the possibility of cross-overs. We illustrate by considering different flavors of *volatile* variables.

4.1 JLS 2 volatiles

The informal requirement for JLS 2 volatiles is that the read of a variable x by a step s must be answered by a step t ordered before s. This can be formalized in RAO as follows. First, we distinguish between raw variables and volatile variables in the model: the underlying set V of variables is partitioned into V_r (the subset of raw variables) and V_v (the subset of volatile variables). An additional restriction is introduced on the applicability of transformations to volatile variables:

> **JLS 2 Volatility Condition:** LI may not be used to link volatile variables.

Therefore the only way to connect a write by a step s to a read by a different, unordered step t is to use AU to *hb*-order s before t, and use CO to compose the steps.

EXAMPLE 21 (Fig 21) Consider the AO process:

```
v=0; (v=1 | r1=v; r2=v)
```

If `v` is not volatile this process may exhibit `r1==1,r2==0`:

```
s0:v=0;(s1:v=1|s2:r1=v;s3:r2=v)
s0:v=0;(s1:v=1|s2:r1=v;s3:r2=0)(LI, s0->s3)
s0:v=0;(s1:v=1|s2:r1=1;s3:r2=1)(LI, s1->s2)
```

However, if `v` is volatile the application of LI is not permitted. For `r1` to read `1`, `s1` must be ordered before `s2`. And it must lie after `s0`. But this will force `r2==1`. □

However, JLS 2 volatiles do not guarantee reliable visibility of writes to raw variables through a volatile write/read pair.

EXAMPLE 22 (Fig 8) Consider the AO process, with `v` volatile:

```
x=0; v=false; (x=1; v=true | r1=v; r2=r1?x )
```

It is desired that if the write to `r2` executes, it writes `1`. That is, a write on a raw variable `x` can be communicated reliably through the synchronization offered by the write to the volatile variable `v`.

Unfortunately, this behavior is not guaranteed. For instance:

```
x=0;v=false;(x,v=1,true|r1=v;r2=r1?x)(CO)
x=0;v=false;(v=true;x=1|r1=v;r2=r1?x)(DE)
x=0;v=false;(v=true;r1=v;r2=r1?x;x=1)(AU, AU)
x=0;v=false;(v,r1,r2=true,true,x;x=1)(CO,CO,CO)
x,v,r1,r2=0,true,true,0;x=1          (CO,CO)
x,v,r1,r2=1,true,true,0              (CO)
```

□

Examples also demonstrate that JLS 2 volatiles do not satisfy the Fundamental Property—see full version of the paper.

4.2 DX-restricted Volatiles

The root cause of this problem is that writes to raw variables are permitted to be reordered with writes to volatile variables. This can be prevented in RAO by requiring in addition to the condition in the previous section:

> DX **Restriction:** DX may not be used to decompose f if f reads or writes a volatile variable.

EXAMPLE 23 (Fig 8, revisited) Consider the AO process, with `v` volatile:

```
x=0;v=false;(x=1;v=true|r1=v;r2=r1?x)
```

Now the desired behavior (if the write to `r2` executes, it writes `1`) can be guaranteed. The only way for `r1=v` to see `v=true` is through an AU (preceded optionally by a CO of `x=1` and `v=true`), followed by a CO. But then it must be the case that `x=1` *hb* `r2=r1?x` (or `x,v=1,true` *hb* `r2=r1?x`), and the desired behavior is guaranteed. □

4.3 JLS 3 volatiles

EXAMPLE 24 (Fig 22) Consider the process:

```
 x=0; y=0; v=0;
(r1=x;v=0;r2=v;y=1 | r3=y;v=0;r4=v;x=1)
```

where only the variable `v` is volatile. The model permits the behavior `r1==r3==1` per the following derivation:

```
x=0;y=0;v=0;
(r1=1;v=0;r2=v;y=1|r3=1;v=0;r4=v;x=1)(LI,LI)
x=0;y=0;v=0;
(r1,v,r2,y=1,0,0,1|r3,v,r4,x=1,0,0,1)   (CO*)
```

□

The resulting process is a completed execution, with a cross-over. Note that all the reads of the volatile variable v have not been totally ordered in the above example. The JLS 3[6] design for volatiles solves this problem by requiring a a total *synchronization order* (SO) on all reads and writes of volatile variable x. Further, there is required to be an *hb*-edge between a write of a volatile variable x and *all* SO-subsequent reads of x.

Formally, this requirement is implemented in RAO exactly as stated above. In addition to the requirements of the previous two sections, we redefine the notion of completed execution as follows:

> **JLS 3 Volatility Condition:** An AO process is a completed execution iff all its steps are completed and there exists a total order on all steps that read or write volatile variables (the *synchronization order*, SO) and there is an *hb*-edge between a write of a volatile variable x and all SO-subsequent reads of x.

Modulo this change, the notion of SC execution is unchanged from Definition 8. To satisfy this requirement *hb*-edges may need to be added, using AU. [5] With these conditions all three Test Cases—(21, 22 and 24)—are satisfied.

EXAMPLE 25 (Fig 22, revisited) Consider the process:

```
x=0;y=0;v=0;(r1=x;v=0;r2=v;y=1|r3=y;v=0;r4=v;x=1)
```

where only v is volatile. Consider:

```
 x=0;y=0;v=0;
(s0: r1=x; v1: v=0; v2: r2=v; s1: y=1
 | s2: r3=y; v3: v=0; v4: r4=v; s3: x=1)
x=0;y=0;v=0;
(s0: r1=1; v1: v=0; v2: r2=v; s1: y=1
 |s2: r3=1; v3: v=0; v4: r4=v; s3: x=1)
                        (LI s3->s0,s1->s2)
```

The resulting process is not a completed execution. There must be a total synchronization order on the steps v1,v2,v3,v4 satisfying the desired condition. Either v2 must lie after v3 or v4 must lie after v1. Any attempts to add *hb*-edges to satisfy the condition above will result in the conditions for one of the links to be violated: the target of a link will be *hb* its source. Therefore it is not possible to complete this process. □

4.4 Main theorem

Let Vl range over the definitions of volatiles (excluding JLS 2, which does not satisfy the Fundamental Property, as discussed above). Let the notion of an RAO(Vl) (AO(Vl)) model stand for the notion of an RAO (AO) model on top of a set of variables which are partitioned into raw and volatile variables, and for which the application of transformations on volatile variables is restricted per Vl, and the definition of completed execution is changed (if necessary) as per Vl. The following results carry over from AO.

LEMMA 20. *For all* AO(Vl) *processes P, Q if P is well-behaved and $P \xrightarrow{X} Q$ then:*

Good behavior is $\xrightarrow{X}$-invariant. *Q is well-behaved.*

SC behavior is $\xrightarrow{X}$-invariant. $\mathrm{sc}(Q) \subseteq \mathrm{sc}(P)$.

IO behavior is $\xrightarrow{X}$-invariant. $\mathrm{io}(Q) \subseteq \mathrm{sc}(P)$.

THEOREM 21 (FUNDAMENTAL PROPERTY). *Let P be a well-behaved* AO(Vl) *process. Then* $\mathrm{io}(P) \subseteq \mathrm{sc}(P)$.

5. Conclusion and future work

We believe this paper is a first step towards establishing a systematic understanding of weak memory models.

We anticipate three major ways in which the work presented in this paper will be extended in future work. First, a programming language designer may wish to use a more "localized" notion of visibility of steps, replacing the global *hb* partial order with more refined "per processor" partial orders [9]. We believe the results in this paper can be extended to cover this case. Second, we believe it will be possible to develop a parametrized family of models RAO(Ξ), where Ξ is a family of SC transformations satisfying certain conditions, in such a way that all models in this family satisfy the requirements in Section 1. For instance this would help in treating languages for which the language designer does not wish to permit AU. Third, we believe that different concurrency and synchronization constructs (e.g. X10's async and finish and various flavors of locks and isolated and atomic execution) can be modeled on top of this framework.

We plan to report on these ideas in subsequent work.

Acknowledgements We gratefully acknowledge extended discussions with Doug Lea, Bill Pugh, Jeremy Manson, Allan Kielstra, Vivek Sarkar, Suresh Jagannathan, Tony Hoare, Hans Boehm and the participants of the Java Memory Model list. Comments from PPoPP reviewers were very helpful. Vijay Saraswat, Maged Michael and Christoph von Praun were supported in part by DARPA under contract No. NBCH30390004. Radha Jagadeesan was supported in part by NSF 0430175.

References

[1] Arvind and J.-W. Maessen. Memory model = instruction reording + store atomicity. In *Proceedings of 33d Annual International Symposium on Computer Architecture*, 2006.

[2] H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI*, pages 261–268, 2005.

[3] P. Charles, C. Donawa, C. Grothoff, K. Ebcioglu, A. Kielstra, V. Saraswat, V. Sarkar, and C. von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[4] G. Gao and V. Sarkar. Location Consistency – A New Memory Model and Cache Consistency Protocol. *IEEE Transactions on Computers*, 49(8):798–813, August 2000.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'90)*, pages 15–26, June 1990.

[6] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison Wesley, 2005.

[7] J. Hoeflinger and B. de Supinsky. The OpenMP memory model. June 2005.

[8] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

[9] D. Lea. Alternatives to SC. Message to C++ threads standardization list, Thu Jan 11 2007.

[5] The addition of AU edges may not be possible because of the presence of links. Thus it is possible that starting with an AO process P, there is a sequence of linkings resulting in a process which cannot be completed into an execution. A safe strategy is to first introduce AU edges as needed to satisfy the condition above, and then add LI links.

[10] J. Manson, B. Pugh, and S. Adve. The Java Memory Model. In *POPL '05. Proceedings of the 32d ACM SIGPLAN-SIGACT on Principles of programming languages*, Jan. 2005.

[11] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.

[12] W. Pugh. Java Memory Model Causality Test Cases. Technical report, U Maryland, 2004. On `www.cs.umd.edu`, as `~pugh/java/memoryModel/`.

[13] V. Saraswat. Concurrent Constraint-Based Memory Machines: A Framework for Java Memory Models. In *ASIAN*, pages 494–508, 2004.

[14] V. Saraswat and R. Jagadeesan. Concurrent Clustered Programming. In *Concur*, pages 353–367, 2005.

[15] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programsthat Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[16] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. A. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPOPP*, pages 2–13, 2005.

[17] J. E. Vuillemin. *Proof-techniques for recursive programs.* PhD thesis, 1974.

[18] C. Wallace, G. Tremblay, and J. Amaral. The Tamability of the Location Consistency Memory Model, 2002.

[19] K. Yelick, D. Bonachea, and C. Wallace. A Proposal for a UPC Memory Consistency Model, v1.1. Technical Report LBNL Technical Report (draft), 2004.