Parallel Assertions for Architectures with Weak Memory Models

Daniel Schwartz-Narbonne[1], Georg Weissenbacher[1,2], and Sharad Malik[1]

[1] Princeton University
[2] Vienna University of Technology, Austria

Assertions are a powerful and widely used debugging tool in sequential programs, but are ineffective at detecting concurrency bugs. We recently introduced parallel assertions which solve this problem by providing programmers with a simple and powerful tool to find bugs in parallel programs. However, while modern computer hardware implements weak memory models, the sequentially consistent semantics of parallel assertions prevents these assertions from detecting some feasible bugs. We present a formal semantics for parallel assertions that accounts for the effects of weak memory models. This new formal semantics allows us to prove the correctness of two key optimizations which significantly increase the speed of a runtime assertion checker on a set of PARSEC benchmarks. We discuss the probe effect caused by checking these assertions at runtime, and show how our new semantics allows the detection of bugs that were undetectable in the previous semantics.

1 Introduction

Assertions are a powerful and widely used debugging tool. They allow programmers to state their expectations about program executions, and provide a mechanism to indicate when these expectations are violated. However, while assertions have proven to be a valuable tool for debugging sequential programs, they have fundamental limitations that restrict their utility in parallel programs.

If a programmer asserts a property ϕ in a sequential program, he can safely assume that the code *after* the assertion executes in an environment in which ϕ is true. In a parallel program, however, other threads can interfere and invalidate the property ϕ. The programmer wants to state "while this code is executing, ϕ must hold", but only has a way to say "before this code executes, ϕ holds".

Parallel Assertions [13] provide an elegant solution to this problem. Instead of evaluating an assertion at a single point in time, a parallel assertion is associated with a syntactic scope, delineated with the keywords `thru` and `passert(`ϕ`)`. The assertion ϕ must hold at all times between the begin and end of the scope.

Atomicity violations and order violations comprise the majority of concurrency bugs [11]. These bugs have the same ultimate cause: a read or a write was issued by a thread at a time when it should not have been able to interfere. In order to allow the specification of such non-interference properties, we allow parallel assertions to refer to memory accesses by means of the operators LocalWrite LW(x), RemoteWrite RW(x), LocalRead LR(x), RemoteRead

RR(x), which indicate whether a given variable is read or written by a local or remote thread. The history operator HasOccurred (HO) allows the programmer to make assertions about a (limited) history of the execution. The assertion if Fig. 1, for example, checks whether the variable `x` is initialised by the local thread before it is read by any remote thread. These features make parallel assertions highly expressive, allowing them to capture 14 out of 17 real world bugs from the University of Michigan bug bench [17].

```
thru {
     ... ; x = 1; ...
} passert(! RR(x) || HO(LW(x))
```

Fig. 1. A simple assertion

We recently presented a runtime checker for parallel assertions [12]. This work, however, is limited by the fact that the semantics of parallel assertions in [13] is based on a sequentially consistent (SC) memory model. Modern processor architectures do not satisfy this requirement, and enforcing SC on a weaker memory model using fences slows down the execution and effectively *masks bugs*.

Contributions: We present a formal operational semantics for parallel assertions that accounts for the effects of weak memory models (§2, §3). This model enables two key optimisations (event filtering and relaxed timing) which we present and prove correct in (§4). We discuss the impact of fences in our model in (§5). We implemented a run-time checker for weak memory systems, and show the significant (order 2×) speedups enabled by our optimisations (§6).

2 Observing Program Executions

The evaluation of an assertion is based on the observation of an execution of the program under test. An execution is characterised by a series of events (discussed in §2.1) and the order in which they are observed (§2.2).

2.1 Program Events

A parallel program comprises a set of threads, each of which generates a series of *observable* events. Events are generated by instructions executed by the processor, and each instruction may result in a number of events. The execution of the assignment `x:=y+z`, for instance, may give rise to two read and one write events. We intentionally base the specification of our assertion language on program events rather than on the instructions of the underlying programming language. The rationale for this decision is that the C++ standard [7] does not provide a semantics for programs with race conditions. In practice, though, a compiler would still generate assembly code (albeit with a compiler-specific behaviour) for such a program. It is exactly in such corner cases that parallel assertions enabling the programmer to debug the flawed program are particularly valuable.

We use $\mathbb{E}$ to denote the set of all observable program events. Formally, an event is a tuple comprising a unique identifier *uid*, the thread identifier *tid* of the thread that generated the event, the *type* of the event, and any data associated with that particular type of event. We distinguish the following types of events:

Memory events $\mathbb{M}$ are physical memory accesses, such as reads and writes. A memory event is a tuple $\langle uid, tid, type, \ell, v\rangle$, where $uid, tid \in \mathbb{N}$ are unique identifiers, $type \in \{\text{READ}, \text{WRITE}\}$ represents the direction of the access, ℓ is the memory location accessed, and $v \in \mathbb{V}$ (where $\mathbb{V}$ is a set of values) is the value read or written. We use $W_{tid}\,\ell\,v$ to denote the write access $\langle uid, tid, \text{WRITE}, \ell, v\rangle$ and $R_{tid}\,\ell$ to denote a read access $\langle uid, tid, \text{READ}, \ell, v\rangle$ when uid (and v, respectively) is not relevant in the given context.

Fence (barrier) events $\mathbb{F}$ affect the legal orderings in an execution by enforcing ordering constraints on memory operations issued before and after the fence instruction (c.f. §2.2). A fence event is a tuple comprising a unique identifier *uid*, a thread identifier *tid*, an architecture dependent fence-type *type*, and an optional (architecture-dependent) set of ordering constraints (c.f. §5).

Scope events $\mathbb{S}$ are generated upon entry or exit of a syntactic assertion scope. A scope entry event is a tuple $\langle uid, tid, \text{ENTRY}, \phi_{uid}\rangle$, where ϕ_{uid} is an assertion (defined in §3.1). A scope exit event is a tuple $\langle uid, tid, \text{EXIT}, uid_{\text{ENTRY}}\rangle$, where uid_{ENTRY} represents the unique entry event corresponding to the scope exit.

We use `skip` to denote events that are irrelevant for assertion evaluation.

2.2 Observation Order for Threads

An observation of a program execution is a sequence of program events. From the viewpoint of a thread P_n, each event occurs at a particular point in time (which determines its location in the sequence). In general, there is no global notion of time, and therefore two threads may disagree on the order in which they observe events. Note that we do not distinguish between "thread-local" and "global" events — conceptually, a thread observes *all* events in some order, though in practice it is typically infeasible (and unnecessary) to record all observations.

The following definition (borrowed from [6] and consistent with [1]) determines what constitutes an observation of a memory event — from the point of view of a given thread — in terms of the *local* time of that thread.

Definition 1 (Observability). *A read or write event is observed when the respective memory access takes effect from the point of view of the observer:*

- *A write to a location in memory is said to be observed by an observer P_n when a subsequent read of the location by P_n would return the value written by the write.*
- *A read of a location in memory is said to be observed by an observer P_n when a subsequent write to the location by P_n would have no effect on the value returned by the read.*[3]

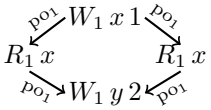
Fence events have no side effect on the execution other than the constraints they impose on the ordering of events (c.f. §3 and §5). Accordingly, *when* a fence event is observed is architecture-dependent and determined by the respective ordering constraints. Therefore, we do not provide a general definition.

[3] This definition is *not* cyclic, since a read observation is defined in terms of the potential *effect* of a write rather than in terms of the observation of a write.

Scope events are only visible to the thread which generates them (see §3). Since they are side-effect free, we assume they are observed by P_n at the point in time when they are generated.

Observations induce a per-thread total order of events, reflecting the order in which they became visible to a particular thread P_n. We represent this order using an irreflexive transitive relation $\xrightarrow{\text{obs}_n} : \mathbb{E} \times \mathbb{E}$. For every pair of events e_1, e_2 and thread P_n, there is a thread-local observation edge $e_1 \xrightarrow{\text{obs}_n} e_2$ iff e_1 is before e_2 in this total order, i.e., the thread observes e_1 before e_2. We use $e_1 \xrightarrow{\text{obs}_n}\!\!\!\!\!\!\times\; e_2$ to abbreviate $\neg(e_1 \xrightarrow{\text{obs}_n} e_2)$ (which implies $e_2 \xrightarrow{\text{obs}_n} e_1$ for $e_1 \neq e_2$).

Note that since $\xrightarrow{\text{obs}_n}$ is irreflexive (i.e., $e_i \xrightarrow{\text{obs}_n}\!\!\!\!\!\!\times\; e_i$) and transitive, cycles are not allowed. Accordingly, executions characterised by a cyclic ordering relation are *infeasible*. While the definition of $\xrightarrow{\text{obs}_n}$ does not impose any further restriction on executions, the program structure imposes certain restrictions as to the order in which events are generated; the thread executing `x:=y+z`, for example, has to perform reads from `y` and `z` before it can issue a write to `x`. We refer to these (thread-local) constraints as the *program order* $\xrightarrow{\text{po}_n} : \mathbb{E} \times \mathbb{E}$. The program order of a thread P_n enforces that certain events are observed in a specific order, i.e., $e_1 \xrightarrow{\text{po}_n} e_2 \Rightarrow e_1 \xrightarrow{\text{obs}_n} e_2$. The relation $\xrightarrow{\text{po}_n}$ corresponds to $\xrightarrow{\text{po}}$ in [2] and to the *sequenced-before* relation of [4] and is determined by the semantics of the language. The diagram to the right, for instance, shows the program order derived from the code fragment `x=1; y=x+x` in the C++ language [7].

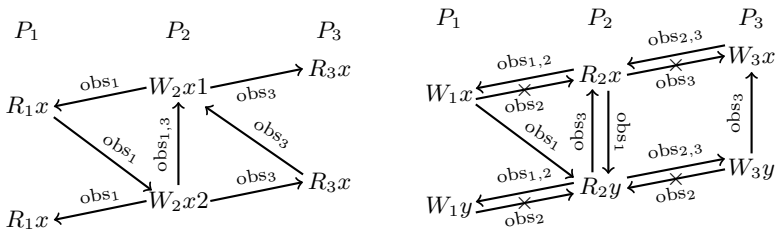


In addition to the restrictions imposed by the semantics of the programming language, we require that events are not reordered across assertion scope boundaries. For events e_{bef} and e_{aft} generated by instructions before and after the beginning of a scope, respectively, and the corresponding scope entry event e_{entry}, we impose $e_{\text{bef}} \xrightarrow{\text{po}_n} e_{\text{entry}} \xrightarrow{\text{po}_n} e_{\text{aft}}$ (and similarly for the end of the scope). We emphasise that the relation $\xrightarrow{\text{obs}_n}$ does not impose ordering constraints on other threads, i.e., $\xrightarrow{\text{obs}_n}$ is *not* global in the sense that $\left(\exists n \,.\, e_i \xrightarrow{\text{obs}_n} e_j\right)$ does not imply $\left(\forall n \,.\, e_i \xrightarrow{\text{obs}_n} e_j\right)$.

The underlying memory model, however, may impose ordering constraints across threads. A common assumption is that the memory model guarantees memory coherence in that for each location, there is a global total order over the writes to that location (c.f. [2]):

$$\left(\exists n \,.\, \langle uid_1, tid_1, \text{WRITE}, \ell, v_1\rangle \xrightarrow{\text{obs}_n} \langle uid_2, tid_2, \text{WRITE}, \ell, v_2\rangle\right) \\ \Rightarrow \left(\forall n \,.\, \langle uid_1, tid_1, \text{WRITE}, \ell, v_1\rangle \xrightarrow{\text{obs}_n} \langle uid_2, tid_2, \text{WRITE}, \ell, v_2\rangle\right) \quad (1)$$

Example 1 (Memory Coherence). Fig. 2(a) depicts an execution invalidated by the coherence constraint (1). Threads P_1 and P_3 observe the write events $W_2\,x\,1$ and $W_2\,x\,2$ in opposite order. By transitivity, $W_2\,x\,2 \xrightarrow{\text{obs}_3} W_2\,x\,1$ follows from



(a) Write serialisation (b) Read order

Fig. 2. P_1 and P_3 observing (a) two instances of W_2x or (b) R_2x and R_2y

$W_2\,x\,2 \xrightarrow{\text{obs}_3} R_3\,x \xrightarrow{\text{obs}_3} W_2\,x\,1$. Similarly, $W_2\,x\,1 \xrightarrow{\text{obs}_1} R_1\,x \xrightarrow{\text{obs}_1} W_2\,x\,2$ implies that $W_2\,x\,1 \xrightarrow{\text{obs}_1} W_2\,x\,2$, and $W_2\,x\,2 \xrightarrow{\text{obs}_1} W_2\,x\,1$ follows from (1) and $W_2\,x\,2 \xrightarrow{\text{obs}_3} W_2\,x\,1$. This contradicts $W_2\,x\,1 \xrightarrow{\text{obs}_1} W_2\,x\,2$. ◁

The coherence constraint (1) also implies that the observation orders for reads of the same location are consistent for all threads. There is, however, no such constraint for write (or read) accesses to *different* locations. Fig. 2(b), for instance, depicts a valid execution in which the threads P_1 and P_3 observe two subsequent reads from x and y in opposite order. We point out that this does *not* contradict the definition in [2, §2.4] that "a read is globally performed as soon as it is performed." This apparent discrepancy stems from the fact that [2] defines when read events are *performed* ("the point when the value of the read is determined" [2, §2.3]) whereas we define when they are *observed*: it is possible to conceive a cache hierarchy in which thread P_n has already *observed* a read while thread P_m may still influence its outcome, i.e., according to Definition 1, a read can be observed before it is actually performed.

3 Operational Semantics of Parallel Assertions

Parallel assertions are evaluated over a given thread's observation of the program state and *execution history*. We provide the syntax and semantics of assertions in §3.1, and subsequently cover executions in §3.2.

3.1 Structural Operational Semantics for Assertions

Each scope event $e \in \mathbb{S}$ has a unique identifier *uid* and an assertion expression $\phi_{uid} \in \mathit{AExpr}$, where *AExpr* is the set of all side-effect-free C++ expressions (defined in [7, §A.4]) augmented with a number of operators (described below). Table 1 shows the (simplified) syntax of assertions in *AExpr*. We hide the complexity of C++ expressions by omitting details about unary and binary operations (*unary-op* and *infix-op*), operator precedence, and type correctness.

assertion ::= HO (*assertion*) |
LR (*lvalue*) | LW (*lvalue*) | RR (*lvalue*) | RW (*lvalue*) |
assertion infix-op asssertion | *unary-op assertion* | *rvalue* | *lvalue*

Table 1. Simplified syntax of assertion expressions

In accordance with the C++ standard [7, §3.10], the non-terminal *lvalue* represents an expression that "designates [...] an object," and an *rvalue* is "a value that is not associated with an object." As in §2.1, we use $v \in \mathbb{V}$ to refer to *rvalue*s and $\ell \in \mathbb{L}$ to refer to *lvalue*s. The access operators LR, LW, RR, and RW in Table 1 take a single *lvalue* as a parameter and check for the occurrence of a memory access to the respective object (c.f. §1). The operator HO takes an assertion as a parameter and returns a Boolean indicating whether this assertion evaluated to true at some point in the respective scope. The use of these operators is demonstrated in Fig. 1 and [13].

We use $\mathit{lvalues}(\mathit{expr}) \subseteq \mathbb{L}$ to denote the set of memory locations (*lvalue*s, respectively) referenced by $\mathit{expr} \in \mathit{AExpr}$ outside an access operator. The operator *lvalues* is defined inductively as follows:

- If $\mathit{lvalues}(\mathit{expr}) = \mathcal{L}$, then $\mathit{lvalues}(\mathrm{HO}(\mathit{expr})) = \mathit{lvalues}(\mathit{unary\text{-}op}\ \mathit{expr}) = \mathcal{L}$.
- Similarly, $\mathit{lvalues}(\mathit{expr}_1\ \mathit{infix\text{-}op}\ \mathit{expr}_2) = \mathit{lvalues}(\mathit{expr}_1) \cup \mathit{lvalues}(\mathit{expr}_2)$.
- If $\mathit{expr} \in \{\mathrm{LR}(\ell), \mathrm{LW}(\ell), \mathrm{RR}(\ell), \mathrm{RW}(\ell)\}, \ell \in \mathbb{L}$, then $\mathit{lvalues}(\mathit{expr}) = \emptyset$.
- Finally, $\mathit{lvalues}(\ell) = \{\ell\}$ for $\ell \in \mathbb{L}$ and $\mathit{lvalues}(v) = \emptyset$ for $v \in \mathbb{V}$.

Similarly, $\mathit{accessops}(\mathit{expr}) \subseteq \mathit{AExpr}$ is the set of all sub-expressions of $\mathit{expr} \in \mathit{AExpr}$ of the form $\mathrm{LR}(\ell)$, $\mathrm{LW}(\ell)$, $\mathrm{RR}(\ell)$, or $\mathrm{RW}(\ell)$ (where $\ell \in \mathbb{L}$). Intuitively, $\mathit{lvalues}(\mathit{expr})$ represents all memory locations whose value is relevant to the evaluation of *expr*, and $\mathit{accessops}(\mathit{expr})$ represents all access events in an expression.

An *assertion set* α is a set of *tagged* assertion expressions $\mathit{uid} : \mathit{expr}$ (where $\mathit{expr} \in \mathit{AExpr}$). Each set α can be partitioned into sets $\alpha|_{\mathit{uid}}$, which denotes the restriction of α to elements tagged with *uid*. The set $\alpha|_{\mathit{uid}}$ itself is inductively defined for each *uid* as the smallest set satisfying the following rules:

- The assertion $\mathit{uid} : \phi_{\mathit{uid}}$ as well as its negation $\mathit{uid} : (\neg\phi_{\mathit{uid}})$ are in $\alpha|_{\mathit{uid}}$.
- If $\mathit{uid} : (\neg \mathit{expr}) \in \alpha|_{\mathit{uid}}$, then $\mathit{uid} : \mathit{expr} \in \alpha|_{\mathit{uid}}$.
- If $\mathit{uid} : (\mathit{expr}_1\ \mathit{bop}\ \mathit{expr}_2) \in \alpha|_{\mathit{uid}}$, then $\mathit{uid} : \mathit{expr}_1$ and $\mathit{uid} : \mathit{expr}_2$ are in $\alpha|_{\mathit{uid}}$.
- If $\mathit{uid} : (\mathrm{HO}(\mathit{expr})) \in \alpha|_{\mathit{uid}}$, then $\mathit{uid} : \mathit{expr} \in \alpha|_{\mathit{uid}}$

Here, $\mathit{expr}, \mathit{expr}_1, \mathit{expr}_2 \in \mathit{AExpr}$ and *bop* represents the Boolean connectives supported by the programming language (e.g., $\wedge$ or $\vee$). Intuitively, $\alpha|_{\mathit{uid}}$ contains the assertion ϕ_{uid} and its negation $\neg\phi_{\mathit{uid}}$, as well as all sub-expressions of ϕ_{uid}. $\mathcal{A}$ denotes the set of all conceivable assertion sets.

Example 2. The assertion set α for $!\,\mathrm{RR}(x) || \,\mathrm{HO}(\mathrm{LW}(x))$ (from Fig. 1) comprises the original assertion as well as the elements $\mathit{uid} : !(!\,\mathrm{RR}(x) || \,\mathrm{HO}(\mathrm{LW}(x)))$, $\mathit{uid} : !\,\mathrm{RR}(x)$, $\mathit{uid} : \mathrm{RR}(x)$, $\mathit{uid} : \mathrm{HO}(\mathrm{LW}(x))$, and $\mathit{uid} : \mathrm{LW}(x)$. ◁

A *state* σ is a finite mapping from locations $\mathbb{L}$ to values $\mathbb{V}$. $\mathcal{S}$ denotes the set of all conceivable states, and $\sigma[\ell \mapsto v]$ denotes the state that maps $\ell \in \mathbb{L}$ to $v \in \mathbb{V}$ and all other locations $\ell' \neq \ell$ to $\sigma[\ell']$. For a given set of locations $\mathcal{L} \subseteq \mathbb{L}$

the projection $\sigma|_{\mathcal{L}}$ of σ to $\mathcal{L}$ is the state that maps locations $\ell \in \mathcal{L}$ to $\sigma[\ell]$ and is undefined for all other locations.

A *history* χ is a tuple $\langle \delta, \theta \rangle$[4] of sets of assertion expressions which represents past evaluations of assertions by recording assertion expressions that evaluate to true at some point during the execution. $\chi.\delta$ (of type *AExpr*) represents the *immediate* past reflecting only the most recent event. $\chi.\theta$ is an assertion set representing the distant past, cumulating all tagged assertions that evaluated to true at some point in the past of the current execution trace. $\mathcal{H}$ denotes the set of all conceivable histories.

A *configuration* κ is a tuple $\langle \sigma, \chi, \alpha \rangle$ comprising a state $\sigma \in \mathcal{S}$, a history $\chi \in \mathcal{H}$, and an assertion set $\alpha \in \mathcal{A}$. $\mathcal{C}$ denotes the set of all configurations.

Evaluating Assertions. Assertions are evaluated over a given configuration. We introduce a reduction relation $\rightarrow_a \subseteq \mathcal{C} \times (\mathbb{N} \times \mathit{AExpr}) \times (\mathbb{N} \times \mathit{AExpr})$ for assertions. We use $(\rightarrow_a)^*$ to denote the reflexive transitive closure of $\rightarrow_a$.

1. Assertion expressions (or sub-expressions) that do *not* contain the operators LR, LW, RR, RW, and HO are evaluated over σ according to the semantics of the C++ language. Therefore, $\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \rightarrow_a \mathit{uid} : v$ if *expr* evaluates to $v \in \mathbb{V}$ in state σ.
2. Expressions involving the access operators LR, LW, RR, or RW are evaluated according to the immediate history $\chi.\delta$:

$$\frac{\mathit{expr} \in \chi.\delta}{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \rightarrow_a \mathit{uid} : \mathsf{T}} \quad \mathit{expr} \in \left\{ \begin{array}{l} \mathrm{LR}(\ell),\ \mathrm{LW}(\ell), \\ \mathrm{RR}(\ell),\ \mathrm{RW}(\ell) \end{array} \right\} \tag{2}$$

   and similarly $\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \rightarrow_a \mathit{uid} : \mathsf{F}$ if $\mathit{expr} \notin \chi.\delta$.
3. The operator HO maps its parameter *expr* to true if *expr* evaluates to true in the current configuration, or if *expr* was true at some point in the past.

$$\frac{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \quad (\rightarrow_a)^* \quad \mathit{uid} : \mathsf{T}}{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathrm{HO}(\mathit{expr}) \rightarrow_a \mathit{uid} : \mathsf{T}} \tag{3}$$

$$\frac{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathit{expr} \quad (\rightarrow_a)^* \quad \mathit{uid} : \mathsf{F}}{\langle \sigma, \chi, \alpha \rangle \vdash \mathit{uid} : \mathrm{HO}(\mathit{expr}) \rightarrow_a \mathit{uid} : b} \quad b \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathsf{T} \text{ if } ((\mathit{uid} : \mathit{expr}) \in \chi.\theta) \\ \mathsf{F} \text{ otherwise} \end{array} \right. \tag{4}$$

   Note that the parameters of HO must not be reduced by $\rightarrow_a$ or evaluated over σ unless this step yields T. This is necessary to avoid mixing values of σ and values of past states during the evaluation.

Example 3. The configuration $\kappa \stackrel{\text{def}}{=} \langle \sigma, \langle \{\mathrm{LR}(x)\}, \{\mathit{uid} : \mathrm{LW}(x)\} \rangle, \alpha \rangle$ (with α as in Example 2) reflects a recent read access as well as a previous write access to x by the asserting thread. Thus, $\rightarrow_a$ yields T for $\mathrm{HO}(\mathrm{LW}(x))$ (by rule 4, since $\kappa \vdash \mathit{uid} : \mathrm{LW}(x) \rightarrow_a \mathit{uid} : \mathsf{F}$ and $\mathit{uid} : \mathrm{LW}(x) \in \chi.\theta$) and $!\mathsf{F}$ for $!\,\mathrm{RR}(x)$ (since $\mathrm{RR}(x) \notin \chi.\delta$). The assertion $!\,\mathrm{RR}(x) || \,\mathrm{HO}(\mathrm{LW}(x))$ does not fail at this point. $\lhd$

[4] Our notation is inspired by the Dirac delta function δ and the Heaviside function θ.

$$\frac{}{\langle (W_{tid}\,\ell\,v) :: ex,\ \langle \sigma,\chi,\alpha\rangle\rangle \ \rightarrow\ \langle ex,\ \langle \sigma[\ell \mapsto v], \langle \{\mathrm{LW}(\ell)\}, \chi'.\theta\rangle, \alpha\rangle\rangle} \quad tid = n \qquad (6)$$

$$\frac{}{\langle (R_{tid}\,\ell) :: ex,\ \langle \sigma,\chi,\alpha\rangle\rangle \ \rightarrow\ \langle ex,\ \langle \sigma, \langle \{\mathrm{LR}(\ell)\}, \chi'.\theta\rangle, \alpha\rangle\rangle} \quad tid = n \qquad (7)$$

$$\frac{}{\langle (W_{tid}\,\ell\,v) :: ex,\ \langle \sigma,\chi,\alpha\rangle\rangle \ \rightarrow\ \langle ex,\ \langle \sigma[\ell \mapsto v], \langle \{\mathrm{RW}(\ell)\}, \chi'.\theta\rangle, \alpha\rangle\rangle} \quad tid \neq n \qquad (8)$$

$$\frac{}{\langle (R_{tid}\,\ell) :: ex,\ \langle \sigma,\chi,\alpha\rangle\rangle \ \rightarrow\ \langle ex,\ \langle \sigma, \langle \{\mathrm{RR}(\ell)\}, \chi'.\theta\rangle, \alpha\rangle\rangle} \quad tid \neq n \qquad (9)$$

Fig. 3. Reduction rules for read and write accesses

3.2 Operational Semantics for Events

Given a set of events $\mathbb{E}$, its Kleene closure $\mathbb{E}^*$ is the set of all sequences of events in $\mathbb{E}$, including the empty sequence ϵ. We use the ML-like notation :: for sequence concatenation. An execution of a thread with $tid = n$ is a sequence $ex \in \mathbb{E}^*$ of events such that for every sub-sequence $e_i :: e_{i+1}$ of ex we have $e_i \xrightarrow{\mathrm{obs}_n} e_{i+1}$. The semantics of an execution is determined by the (reflexive) reduction relation $\rightarrow \subseteq (\mathbb{E}^* \times \mathcal{C}) \times (\mathbb{E}^* \times \mathcal{C})$ which characterises the impact of events on configurations. In the following, we define this reduction relation $\rightarrow$ based on $\rightarrow_a$.

A common side effect of all events is the modification of the history. Let $\langle \sigma, \chi, \alpha\rangle$ be the current configuration and let χ and χ' denote the history before and after an event, respectively. For all events, $\chi'.\theta$ is $\chi.\theta$ augmented with all assertion expressions in α that evaluate to true in the previous configuration:

$$\chi'.\theta \stackrel{\mathrm{def}}{=} \chi.\theta \cup \{(uid : expr) \in \alpha \mid \langle \sigma,\chi,\alpha\rangle \vdash (uid : expr)\ (\rightarrow_a)^*\ uid : \mathsf{T}\} \qquad (5)$$

The reduction rules presented in the following refer to $\chi'.\theta$ as defined in (5). From now on, we use n to denote the identifier of the current (asserting) thread.

1. Fig. 3 shows the reduction rules for memory events. We distinguish between events generated by thread n and events generated by other threads in order to determine their effect on the immediate past $\chi.\delta$.
2. Fig. 4 shows the reduction rules for scope events. Upon entry of a scope, the respective assertion is added to the assertion set. Note that Rule (11) does not allow us to exit a scope if the corresponding assertion (identified by *eid*) failed. Finally, a thread only observes the scope events generated by itself.
3. Skip and fence events modify the history in accordance with (5). In addition, the memory model (see §5) may impose ordering constraints on fence events.

$$\frac{}{\langle e_i :: ex,\ \langle \sigma,\chi,\alpha\rangle\rangle \ \rightarrow\ \langle ex,\ \langle \sigma, \langle \emptyset, \chi'.\theta\rangle, \alpha\rangle\rangle} \quad e_i \in (\mathbb{F} \cup \{\texttt{skip}\}) \qquad (12)$$

$$\frac{}{\langle\langle uid, tid, \text{ENTRY}, \phi_{uid}\rangle :: ex,\ \langle\sigma, \chi, \alpha\rangle\rangle \ \rightarrow\ \langle ex,\ \langle\sigma, \langle\emptyset, \chi'.\theta\rangle, \alpha \cup \alpha|_{uid}\rangle\rangle} \quad tid = n \tag{10}$$

$$\frac{(eid : \neg\phi_{eid}) \notin \chi.\theta\ \wedge\ \langle\sigma, \chi, \alpha\rangle \vdash eid : \phi_{eid}\,(\rightarrow_a)^*\ eid : \mathsf{T}}{\langle\langle uid, tid, \text{EXIT}, eid\rangle :: ex,\ \langle\sigma, \chi, \alpha\rangle\rangle \ \rightarrow\ \langle ex,\ \langle\sigma, \langle\emptyset, \chi'.\theta\rangle, \alpha \setminus \alpha|_{eid}\rangle\rangle} \quad tid = n \tag{11}$$

Fig. 4. Reduction rules for scope events

Assertion Failures. In case of a failure of an active assertion ϕ_{uid}, the configuration can be reduced to a (canonical) error configuration **error**. We allow this rule to be applied *non-deterministically* at any point after an assertion failure (but at latest upon exit of the corresponding scope, see Rule (11)). This leaves some freedom for the implementation as to when a failed assertion is reported.

$$\frac{(uid : \neg\phi_{uid}) \in \chi.\theta\ \vee\ (uid : \phi_{uid}) \in \alpha\ \wedge\ \kappa \vdash (uid : \phi_{uid})\,(\rightarrow_a)^*\,(uid : \mathsf{F})}{\langle ex,\ \kappa\rangle\ \rightarrow\ \textbf{error}} \tag{13}$$

Example 4. Consider an execution $W_n\,x\,1 :: R_{tid}\,x$ (where $tid \neq n$) starting in the configuration $\kappa \stackrel{\text{def}}{=} \langle\sigma, \langle\{\text{LR}(x)\}, \{uid : \text{LW}(x)\}\rangle, \alpha\rangle$ introduced in Example 3. By rule 6, we derive $\langle R_{tid}\,x, \langle\sigma[x \mapsto 1], \langle\{\text{LW}(x)\}, \{uid : \text{LW}(x)\}, \alpha\rangle\rangle$ from $\langle W_n\,x\,1 :: R_{tid}\,x, \kappa\rangle$. Note that $uid : \text{LR}(x)$ is *not* added to $\chi'.\theta$ by rule 5, since it is not an element of the assertion set α. ◁

4 Optimisations

In this section, we formally prove that only a fraction of the events in an execution is necessary to evaluate an assertion. Since logging information can be expensive, there is a significant optimisation opportunity in filtering the executions before they are evaluated. In particular, this means that throughout a scope we can dismiss the events that are irrelevant to the corresponding assertion, as long as assertion failures are preserved:

Definition 2. *Let* $\text{ex}_1, \text{ex}_2 \in \mathbb{E}^*$ *be two executions delimited by a scope with assertion* ϕ_{uid} *that contains no other scope events. Then,* ex_1 *and* ex_2 *are* parallel assertion equivalent *with respect to* ϕ_{uid} *iff in all configurations* $\langle\sigma, \chi, \alpha|_{\text{uid}}\rangle$ *it holds that* $(\langle\text{ex}_1,\ \langle\sigma, \chi, \alpha|_{\text{uid}}\rangle\rangle\,(\rightarrow)^*\,\textbf{error}) \Leftrightarrow (\langle\text{ex}_2,\ \langle\sigma, \chi, \alpha|_{\text{uid}}\rangle\rangle\,(\rightarrow)^*\,\textbf{error})$.

Nested Assertion Scopes. For the purpose of checking whether a specific assertion ϕ_{uid} is violated by an execution, we can treat other scope events similar to `skip`. Scope events occurring in threads other than the current one have no impact on the evaluation of ϕ_{uid} (c.f. Rules 10, 11). A nested scope event only affects the execution if its respective assertion fails. Therefore, it is legal to process assertion scopes independently as long as we report the first assertion that fails

(upon exit of the corresponding scope). This allows us to define parallel assertion equivalence (Definition 2) with respect to a *single* parallel assertion.

In the following, we formally define the projection of configurations and executions to a given assertion and prove that projection preserves assertion failures.

Definition 3. *We define the projection of a configuration* $\langle \sigma, \chi, \alpha \rangle$ *to a given assertion* ϕ_{uid} *as* $\langle \sigma, \chi, \alpha \rangle|_{\phi_{\text{uid}}} \stackrel{\text{def}}{=} \langle \sigma|_{\text{lvalues}(\phi_{\text{uid}})}, \langle \chi.\delta \cap \text{accessops}(\phi_{\text{uid}}), \chi.\theta \rangle, \alpha \rangle$.

Theorem 1. *Projecting a configuration* $\langle \sigma, \chi, \alpha \rangle$ *to an assertion* ϕ_{uid} *has no impact on the evaluation of* ϕ_{uid}*:*

$$\forall \kappa \in \mathcal{C}\,.\,\forall \text{expr}_1 \in \alpha|_{\text{uid}},\, \text{expr}_2 \in \text{AExpr}\,.\\ (\kappa \vdash \text{expr}_1\ (\rightarrow_a)^*\ \text{expr}_2) \Leftrightarrow (\kappa|_{\phi_{\text{uid}}} \vdash \text{expr}_1\ (\rightarrow_a)^*\ \text{expr}_2)$$

Proof: By induction on the structure and height of derivations generated by the reduction rules in §3.1. ∎

Definition 4. *The projection of an execution* $\text{ex} \in \mathbb{E}^*$ *to an assertion* ϕ_{uid} *is defined inductively by* $\epsilon|_{\phi_{\text{uid}}} \stackrel{\text{def}}{=} \epsilon$ *and* $(e :: \text{ex})|_{\phi_{\text{uid}}} \stackrel{\text{def}}{=} e|_{\phi_{\text{uid}}} :: (\text{ex}|_{\phi_{\text{uid}}})$*, where*

$$e|_{\phi_{\text{uid}}} \stackrel{\text{def}}{=} \begin{cases} e & \textit{if } (e = W_{\text{tid}}\,\ell\,v) \wedge \\ & \quad \left(\begin{array}{r} (\ell \in \text{lvalues}(\phi_{\text{uid}})) \quad \vee \\ (\text{tid} = n \wedge \text{LW}(\ell) \in \text{accessops}(\phi_{\text{uid}})) \vee \\ (\text{tid} \neq n \wedge \text{RW}(\ell) \in \text{accessops}(\phi_{\text{uid}})) \end{array} \right) \\ & \textit{or } (e = R_{\text{tid}}\,\ell v) \wedge \\ & \quad \left(\begin{array}{l} (\text{tid} = n \wedge \text{LR}(\ell) \in \text{accessops}(\phi_{\text{uid}})) \vee \\ (\text{tid} \neq n \wedge \text{RR}(\ell) \in \text{accessops}(\phi_{\text{uid}})) \end{array} \right) \\ \texttt{skip} & \textit{otherwise} \end{cases}$$

Theorem 2. *Let* $\text{ex} \in \mathbb{E}^*$ *be an execution that is delimited by a scope associated with* ϕ_{uid} *and does not contain any other assertion scopes. Then* ex *and* $\text{ex}|_{\phi_{\text{uid}}}$ *are parallel assertion equivalent with respect to the assertion* ϕ_{uid}*.*

The proof of Theorem 2, led by induction over the length of *ex*, shows that events can be treated as `skip` as long they have no impact on the evaluation of ϕ_{uid} (in accordance with Theorem 1).

Theorems 1 and 2 enable the following optimisation. For each scope instance, no events need to be logged before its respective entry event. Upon reaching a scope entry event with assertion ϕ_{uid}, our implementation records only the relevant fraction $\sigma|_{\phi_{uid}}$ of the state. After that, it is sufficient to log all events $e|_{\phi_{uid}}$ (in observation order) until the corresponding scope exit event is reached.

Relaxed Order Observations. Under certain conditions it is sufficient to approximate the observation order $\xrightarrow{\text{obs}_n}$ by a partial order. In particular, we show that for a certain class of assertions all that matters is the order of events with respect to the scope events $\mathbb{S}$.

Theorem 3. *Let* ex *be an execution that is delimited by a scope associated with* ϕ_{uid} *and does not contain any other assertion scopes, and let* $\pi(\text{ex})$ *be an arbitrary permutation of* ex*. If one of the following conditions holds for* ϕ_{uid}*, then* ex *and* $\pi(\text{ex})$ *are parallel assertion equivalent with respect to* ϕ_{uid}*:*

i ϕ_{uid} *does not contain the operator* HO *and* $\text{lvalues}(\phi_{\text{uid}}) = \emptyset$*.*
ii ϕ_{uid} *does not contain the operator* HO*,* $\text{lvalues}(\phi_{\text{uid}}) = \{\ell\}$ *(for some* $\ell \in \mathbb{L}$*),* *and* $\text{accessops}(\phi_{\text{uid}}) = \emptyset$*.*

Proof: We prove that the order of the sequence is irrelevant by showing that the following logical equivalence holds:

$$(\langle ex, \langle \sigma, \chi, \alpha|_{uid}\rangle\rangle\,(\rightarrow)^*\,\textbf{error}) \quad \Leftrightarrow$$
$$\exists e \in ex\,.\,\forall \langle \sigma', \chi', \alpha|_{uid}\rangle\,.\,\langle e :: \epsilon, \langle \sigma', \chi', \alpha|_{uid}\rangle\rangle\,(\rightarrow)^*\,\textbf{error}$$

Note that the implication holds trivially in one direction ($\Leftarrow$). For the other direction, we need to show that ϕ_{uid} does not depend on the configuration before the execution of $e \in ex$. Let $\langle \sigma'', \chi'', \alpha|_{uid}\rangle$ be the configuration after the execution of e. By Theorem 1 we have $\langle \sigma'', \chi'', \alpha|_{uid}\rangle\,(\rightarrow_a)^*\,\textbf{error}$ if $\langle \sigma'', \chi'', \alpha|_{uid}\rangle|_{\phi_{uid}}\,(\rightarrow_a)^*\,\textbf{error}$. In case (i), the domain of $\sigma|_{\phi_{uid}}$ is empty and ϕ_{uid} refers only to $\chi''.\delta$. In case (ii), $\sigma|_{\phi_{uid}}$ is only defined for ℓ, and $\chi''.\delta \cap accessops(\phi_{uid}) = \emptyset$. Accordingly, ϕ_{uid} depends on only a single item in the configuration, which can only be updated *atomically* by the events in *ex*. ∎

The conditions for ϕ_{uid} in Theorem 3 are not tight. Other criteria (based on partial order reduction, for instance) may allow for more aggressive optimisations. Note that in the most extreme case — if we can establish statically that the assertion holds — it is not necessary to log any events at all. In general, an approach based in this insight is obviously impractical. However, since observing events and orderings between events is expensive, an improved relaxation function which limits the number of required observations can significantly reduce the work required by the checker.

5 Memory Models and Fences

On a platform that guarantees sequential consistency (SC) the operations of each individual thread are globally observed in a sequential order consistent with the program order. For performance reasons, modern processors do not provide such a guarantee. They do, however, provide *fence* instructions, which enable the programmer (or compiler) to enforce a global ordering between events.

Different platforms provide different types of fences, and their semantics depends on the specific architecture (c.f. [2, 6]). In general, we distinguish between *non-cumulative* and *cumulative* fences. Intuitively, non-cumulative fences prevent thread-local reordering of events across the fence event. Cumulative barriers also affect the order of events of other threads (e.g., by flushing store buffers or caches). One non-cumulative fence operations is `mfence` on Intel processors: it

"guarantees that every load and store instruction that precedes in program order the `mfence` instruction is globally visible before any load or store instruction that follows the `mfence` instruction is globally visible." [1, pg. 4-23][5]

Formally, two memory events $e_1, e_2 \in \mathbb{M}$ occurring before and after an `mfence` event in program order, respectively, will be observed in this order by all threads:

$$\exists n\,.\,\left(e_1 \xrightarrow{\mathrm{po}_n} \texttt{mfence} \xrightarrow{\mathrm{po}_n} e_2\right) \quad \Rightarrow \quad \forall n\,.\,e_1 \xrightarrow{\mathrm{obs}_n} e_2 \tag{14}$$

The `hwsync` instruction of the Power architecture provides a similar guarantee, but is also cumulative in the sense that it separates the memory events *observed* by the thread before and after executing `hwsync`.

Example 5. The assertion in the program in Fig. 5 holds on any platform that guarantees sequential consistency, but may fail on a platform that permits reordering of write events (as shown in the diagram in Fig. 5). A fence between the events $W_1\,x\,1$ and $W_1\,y\,1$ (indicated by a dashed arrow) rules out the failing execution and restores sequential consistency for this program. ◁

The side effect described in Example 5 is not always desirable. If the fence instruction is part of the instrumentation code required to log the write events, then this modification effectively eliminates an erroneous execution. Logging mechanisms requiring stronger guarantees (as provided by the atomic compare-and-swap instruction `lock cmpxchg` on Intel architectures, for instance, which is frequently used to implement locks) exacerbate the probe effect.

A complete formalisation of fences exceeds the scope of our paper; we refer the reader to [2] instead. The following section presents our implementation, which makes use of fences, and discusses their side effects.

6 Implementation

We implemented a runtime checker for parallel assertions called PASSERT [12] as an extension of the LLVM compiler [9]. During compilation, PASSERT instruments read and write accesses in a program annotated with assertions with calls to logging functions. The log is then analysed for assertion violations by a checker (either during or after the execution).

The instrumentation results in a number of side effects. Firstly, logging events takes time, and hence changes the timing behaviour of the program under test. More subtly, the instrumentation adds locks and fences which may rule out executions that are otherwise legal the underlying memory model. In the following, we discuss two key optimisations that dramatically reduced these effects based on the results from §4.

Filtering Optimisation. To counteract the probe effect, we implemented an event filter that conservatively approximates the set of events required by Theorem 2.

Firstly, we use an alias analysis to narrow down which memory locations may affect the evaluation of an assertion, which significantly reduces the number of

[5] The concept of global visibility coincides with our notion of observability.

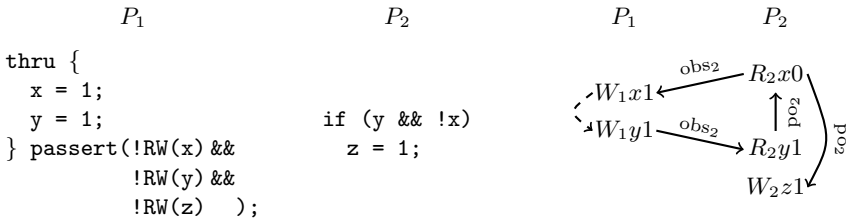


Fig. 5. Restoring sequential consistency using fences (`x` and `y` are initially 0)

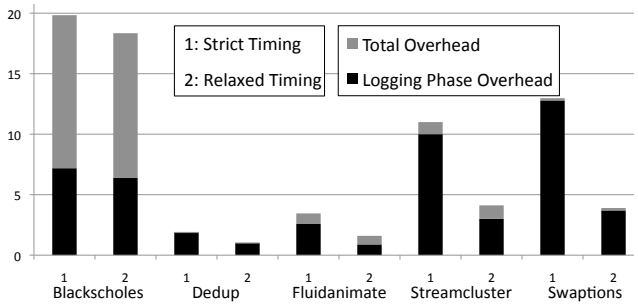


Fig. 6. Runtime overhead for PARSEC benchmarks

instructions that need to be instrumented and therefore eliminates the associated side effect. Secondly, we use dynamic filtering to determine at runtime which locations are not referenced by any live assertions, and hence need not be logged. This further reduces – but does not entirely eliminate – the probe effect.

Filtering effectively *enables* runtime checking. Before this optimisation, most instrumented programs simply ran out of memory, while all our benchmarks completed successfully after filtering.

Time-stamping Mechanisms. Assertion evaluation requires recording the timing of remote events relative to the asserting thread. For events generated by different threads this requires a certain amount of synchronisation. Our implementation uses a global time-stamp for this purpose.

A time-stamping mechanism must correctly associate events with timestamps reflecting the observation order. In a weak memory model, this may need to be enforced through the use of locks and memory fences. Consequently, stronger ordering requirements exacerbate the probe effect, which in turn may rule out otherwise legal executions. In Fig. 5, for example, a time-stamping mechanism inserting a fence between $W_1\,x\,1$ and $W_1\,y\,1$ effectively prevents reordering of these events, thus hiding the bug (as explained in Example 5).

By Theorem 3, the order of events within a scope can be safely ignored in certain cases (such as the program in Example 5). Our experience suggests that

this relaxation is admissible for many assertions: previous work [12] showed that of the 14 out of 17 real world bugs presented in [17] can be captured using parallel assertions; all of those assertions are amenable to relaxed timing. This observation led us to implement two different time-stamping mechanisms:

- *Strict time-stamping* uses locks to guarantee that a shared counter is incremented atomically with the execution of an event, hence providing a total order over all time-stamped events.
- *Smeared time-stamps* yield a partial order sufficiently accurate to evaluate the assertions characterised in Theorem 3. Scope events atomically increment a global counter. All other events e are logged using a preceding and a successive read to the global counter `ts`. $R_{tid}\,\texttt{ts} \xrightarrow{\text{po}_{tid}} e \xrightarrow{\text{po}_{tid}} R_{tid}\,\texttt{ts}$ is enforced using (non-cumulative) fences if necessary, thus avoiding the use of locks.

Smeared time-stamping is sufficient to determine whether an event happened within a certain scope. A potentially ambiguity may arise in the rare case that different time-stamps are recorded before and after the event. If this difference affects the correctness of an assertion, we report a potential error (though we have not observed this in practice).

In order to measure the run-time overhead, we evaluated the runtime overhead of PASSERT by annotating a set of PARSEC benchmarks with parallel assertions. We did not discover any new bugs in this widely used benchmark suite. Strict timing had an overhead of 6.6×, which can be reduced to 3.5× through the use of smeared timestamps (Fig. 6).

7 Related Work

A number different assertion formalisms for parallel programs have been proposed and implemented. JMPaX [14], for instance, checks linear temporal logic properties for Java programs. Phalanx [15] allows the checking of expressive heap assertions in Java programs. SharC [3] enables the static and dynamic verification of rules for sharing individual objects in C. These formalisms address different properties than parallel assertions — a more detailed discussion is given in [13], where we initially introduced parallel assertions.

There is a wide variety of work on debugging programs in the presence of weak memory models. One approach is to minimize the probe effect through hardware based logging. [16] allows the observation of events with minimal (2%) perturbation to the program execution on both TSO(x86) and SC systems. Predictive analyses (e.g. [5]), on the other hand, enable the prediction of possible assertion violations in a weak memory model based on an SC execution. Trace based analysis can also be used to automatically fix bugs. Liu et al. [10], for example, provide a formal semantics for LLVM bytecode under weak memory models, and show how to add fences to prevent erroneous traces. A model checker can exhaustively explore all possible interleavings under a given memory model for all possible inputs (e.g. [8]), and hence capture all bugs albeit at a high computational cost. All of these techniques are orthogonal to our work. Parallel assertions could be implemented as an extension to any of these systems.

8 Conclusion

We provide a formal definition of *parallel assertions*, a novel assertion language for detecting intricate concurrency bugs, and an operational semantics enabling their evaluation on architectures with weak memory models. Unlike the original semantics [13], which assumes sequential consistency, our new semantics is less restrictive and enables the detection of a highly relevant class of bugs introduced by the complexity of modern multi-processor architectures. Secondly, our novel semantics enables two key optimisations which, as demonstrated in §6, are crucial to making run-time checking of parallel assertions feasible.

Acknowledgements. The authors thank Lennart Beringer and the anonymous reviewers for their helpful suggestions and comments.

References

1. Intel 64 and IA-32 architectures software developer's manual, March 2012.
2. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2), 2012.
3. Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: checking data sharing strategies for multithreaded C. In *PLDI*. ACM, 2008.
4. H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*. ACM, 2008.
5. J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *ISSTA*. ACM, 2011.
6. N. Chong and S. Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *MSPC*. ACM, 2008.
7. International Standard 14882 (Programming Languages) C++, Final Committee Draft N3092, March 2010. ISO/IEC.
8. B. Jonsson. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Comput. Archit. News*, 36(5), 2009.
9. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization*. IEEE, 2004.
10. F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*. ACM, 2012.
11. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*. ACM, 2008.
12. D. Schwartz-Narbonne, F. Liu, D. August, and S. Malik. PASSERT: A tool for debugging parallel programs. In *CAV*, LNCS. Springer, 2012.
13. D. Schwartz-Narbonne, F. Liu, T. Pondicherry, D. I. August, and S. Malik. Parallel assertions for debugging parallel programs. In *MEMOCODE*. IEEE, 2011.
14. K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ESEC/FSE*. ACM, 2003.
15. M. Vechev, E. Yahav, and G. Yorsh. Phalanx: parallel checking of expressive heap assertions. In *ISMM*. ACM, 2010.
16. M. Xu, R. Bodik, and M. D. Hill. A hardware memory race recorder for deterministic replay. *IEEE Micro*, 27(1), 2007.
17. J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*. ACM, 2009.