# Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future *

Man Cao Jake Roemer Aritra Sengupta Michael D. Bond

Ohio State University (USA)

{caoma,sengupta,mikebond}@cse.ohio-state.edu, roemer.37@osu.edu

## Abstract

Shared-memory parallel programs are hard to get right. A major challenge is that language and hardware memory models allow unexpected, erroneous behaviors for executions containing data races. Researchers have introduced dynamic analyses that expose weak memory model behaviors, but these approaches cannot expose behaviors due to loading a "future value"—a value written by a program store that executes *after* the program load that uses the value.

This paper presents *prescient memory* (PM), a novel dynamic analysis that exposes behaviors due to future values. PM speculatively returns a future value at a program load, and tries to validate the speculative value at a later store. To enable PM to expose behaviors due to future values in real application executions, we introduce a novel approach that increases the chances of using and successfully validating future values, by profiling and predicting future values and guiding execution. Experiments show that our approach is able to uncover a few previously unknown behaviors due to future values in benchmarked versions of real applications. Overall, PM overcomes a key limitation of existing approaches, broadening the scope of program behaviors that dynamic analyses can expose.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification—reliability; D.2.5 [*Software Engineering*]: Testing and Debugging—monitors, testing tools

***Keywords*** Data races; relaxed memory consistency models; Java memory model; dynamic program analysis

## 1. Introduction

With the widespread adoption of multi-core processors, software must become more parallel in order to scale with successive hardware generations. However, it is notoriously challenging to write and debug shared-memory parallel programs, which can have many possible—and potentially harmful—behaviors that are difficult to reason about. A key problem is that shared-memory languages and architectures provide few, if any, guarantees for *data races*, leading to unexpected, erroneous behaviors. For example, C/C++ executions that are racy (i.e., have a data race) have undefined semantics [2, 11]. Java provides defined but weak semantics for racy executions, in an effort to preserve memory and type safety [34], although later work has shown that this model is impractical to enforce [12, 34, 47].

Data races and their erroneous effects occur nondeterministically and only under certain conditions. Data races are difficult to avoid, find, fix, and eliminate [1, 7, 13, 18–23, 37, 38, 41, 43, 44, 49–51]. Programmers often introduce data races intentionally for performance [9, 10, 28, 29]. Data races and their erroneous effects are thus ubiquitous, even in mature software systems.

Figure 1 shows an example shared-memory program in a Java-like language; `x` and `y` are shared variables, and `r1` and `r2` are locals. Under a weak memory model such as Java's memory model [34], it is legal for both loads to read the value 0, violating the assertion. However, such an outcome would not be possible if memory accesses appeared to execute in their original order (i.e., with *sequential consistency* (SC) semantics [31]). This kind of non-SC behavior not only is permitted in theory, but occurs in practice when compiler and hardware optimizations reorder intra-thread memory accesses.

Since non-SC behaviors tend to manifest infrequently and unexpectedly, researchers have introduced dynamic analyses that intentionally expose non-SC behaviors allowed under weak memory models [17, 24, 29]. However, these dynamic analyses are limited in the kinds of behaviors they can expose. Figure 2 shows an example for which Java's memory model permits both loads to read the value 1. Existing dynamic analyses cannot expose this behavior because they allow loads to read "stale" values (values stored in the past), but not "future" values (values that will be stored in the future).

This paper addresses the challenge of how to expose behaviors due to future values, which (as we explain) is inherently more difficult than for stale values. To the best of our knowledge, we introduce the first dynamic analysis that uses future values and evaluates the effects of using future values in real application executions. Prior work has used future values in the context of model checking of small libraries that use C++ atomic types [40] (Section 9). In contrast to model checking, which explores many executions exhaustively, dynamic analysis faces the challenges of how to expose behaviors due to future values within a single execution—an important goal since model checking techniques generally do not scale to large, long-running programs.

Exposing effects due to future values (as opposed to stale values) presents a unique challenge: if a load reads a future value (i.e., a value that is expected to be stored in the future), this load operation can "change the future," so that the anticipated future value is *no longer* stored in the future, leading to an outcome *not* permitted by weak memory models for safe languages such as Java. For example, suppose a thread's store is control-dependent on the value

* This material is based upon work supported by the National Science Foundation under Grants CSR-1218695, CAREER-1253703, and CCF-1421612.



*ISMM'16*, June 14, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4317-6/16/06...
http://dx.doi.org/10.1145/2926697.2926700

```
Initially x = y = 0

Thread 1:          Thread 2:

y = 1;             x = 1;
r1 = x;            r2 = y;

assert r1 != 0 || r2 != 0
```

**Figure 1.** An assertion failure is possible under Java's memory model. Existing dynamic analyses *can* expose the assertion failure.

```
Initially x = y = 0

Thread 1:          Thread 2:

r1 = x;            r2 = y;
y = 1;             x = 1;

assert r1 == 0 || r2 == 0
```

**Figure 2.** An assertion failure is possible under Java's memory model. Existing dynamic analyses *cannot* expose the failure.

```
Initially x = y = 0

Thread 1:          Thread 2:

r1 = x;            r2 = y;
y = 1;             if (r2 == 0)
                     x = 1;

assert r1 == 0 || r2 == 0
```

**Figure 3.** An assertion failure is *not* possible under Java's memory model. Dynamic analysis must be careful *not* to allow an execution in which the assertion fails.

of a load, as in Figure 3. Then the assertion-violating outcome is impossible under the Java memory model. Thus, special challenges for using future values are (1) how to predict future values that are likely to be stored in the future and (2) how to validate that speculatively used future values are in fact stored in the future.

This paper presents a novel dynamic analysis called *prescient memory* (PM) that exposes behaviors caused by future values, without exposing unjustified future value behaviors.[1] PM achieves this objective by (1) *speculatively* using potential future values at loads and (2) *validating* that the future value is later stored. In order for PM to be useful, it must make reasonably good choices about which *loads* should use future values and which future *values* to use. Our solution to this challenge, which we call the *PM workflow*, consists of three components: (1) profiling of values stored in the future; (2) prediction of which loads should use future values and which future values to use; and (3) deterministic execution from profiling to prediction except when the latter execution unavoidably diverges, which we call *fuzzy replay*. The resulting approach is well suited to exposing (legal) uses of future values in real programs that are too large for exhaustive exploration.

We have implemented PM in a Java virtual machine and evaluated PM's ability to expose erroneous behaviors in benchmarked versions of large, real Java applications. We have also implemented and compared against prior work called *adversarial memory* (AM) [24], which exposes errors due to stale values. PM exposes 7 bugs, i.e., 7 distinct shared variables for which a future leads to erroneous behavior, such as crashes and incorrect output. AM exposes 6 of these 7 bugs (as well as 2 other errors), although in 2 of these cases, PM exposes *different* (and arguably more destructive) erroneous behaviors than AM. These results show that our approach is effective at using future values that can be validated, and that future values can help to expose new bugs and behaviors.

This paper makes the following intellectual and empirical contributions:

- PM is the first dynamic analysis that exposes weak memory model behaviors due to future values in large, real applications. In order to enable PM to expose these behaviors without exhaustive exploration, we introduce a novel approach called the PM workflow that incorporates three components: profiling, prediction, and fuzzy replay. Our evaluation shows that this approach is in fact useful for using future values successfully.
- Our evaluation shows that legal uses of future values exist in real applications and that they can lead to harmful behaviors. Future values alone (i.e., without using any stale values) can often trigger the same bugs that stale values trigger—but the future-value behaviors are sometimes different and more destructive. These results motivate our approach's utility for exposing previously unknown program behaviors.
- An existing line of research shows that seemingly "benign" data races are in fact harmful [17, 24, 28, 29, 39, 40, 45]. By exposing real, destructive behaviors due to future values, our work advances the state of the art in this area.
- Existing language memory models still have difficulty defining what program behaviors should be allowed for an execution with data races [2, 12, 47]. Our approach provides an opportunity to explore this gray area in large, real programs; real-world evidence of controversial examples would inform and influence future language specification revisions.

## 2. Background and Motivation

This section provides background on language memory models and the behaviors that they allow, including behaviors due to "stale" and "future" values. We then motivate both the importance and challenges of exposing behaviors due to future values.

### 2.1 Memory Models

In 1990, Adve and Hill introduced the *data-race-free-0* (DRF0) memory model [3], which guarantees sequential consistency for well-synchronized program executions, i.e., executions that are free of data races. An execution is *sequentially consistent* (SC) if all memory accesses appear to be interleaved in an order that is consistent with each thread's program order [31]. A *data race* occurs when two threads access the same memory location without synchronization, and at least one of the accesses is a store [4]. The rationale for the DRF0 model is that it permits compilers and hardware to perform aggressive intra-thread optimizations, as long as they do not arbitrarily reorder memory accesses across synchronization operations. As long as programmers avoid data races, the effects of optimizations will not be externally visible.

Modern shared-memory languages provide memory models that are based on DRF0 [2, 11, 34]. C and C++ lend undefined semantics to a "racy" execution (i.e., an execution with a data race) [11]. While this situation is acceptable for unsafe languages such as C and C++, preserving memory and type safety in a safe language such as Java demands providing some semantics for racy executions.

The Java memory model (JMM) ensures certain weak semantics for racy program executions [34]. However, subsequent work shows that JMM actually precludes common Java virtual machine (JVM) optimizations [2, 12, 47]. Commercial JVMs thus *violate* JMM, since existing art does not demonstrate how to avoid certain

[1] As we explain, PM exposes future value behaviors even for data-race-free programs (Section 5), but the PM workflow does not (Section 6).

```
Initially x = 0

Thread 1:          Thread 2:

x = 7;             if (x != 0)
                     r2 = r1 / x;
```

**Figure 4.** An example program that can generate a divide-by-zero exception under HBMM.

undesirable results without seriously inhibiting optimizations [12, 47]. Recent work tries to address this issue, using techniques based on event structure models, but it is unclear whether this work perfectly delineates behaviors needed for optimization from other behaviors [27, 42]. Instead of conforming to JMM, JVMs at least conform to a memory model that is the intersection of (1) DRF0 and (2) a weak memory model called the *happens-before memory model* (HBMM), presented next.

***Happens-before memory model (HBMM).*** The following description of HBMM is closely based on prior work [24, 34]. HBMM is an easy-to-understand memory model that provides weak but defined semantics for executions with data races. HBMM limits the values that a load can return according to the *happens-before* relation [30], denoted as $\rightarrow_{hb}$. A load operation $r$ may return the value written by any store $w$ to the same location, if and only if the following properties hold true:

1. $r \not\rightarrow_{hb} w$ (i.e., $w$ happens-before or is concurrent with $r$).
2. There is no intervening store $w'$ to the same memory location such that $w \rightarrow_{hb} w' \rightarrow_{hb} r$.

HBMM thus still permits various behaviors in which memory accesses appear to execute in an order other than program order, i.e., non-SC behaviors. HBMM allows assertion violations in the programs in Figures 1 and 2 (Section 1). As another example, HBMM allows a divide-by-zero exception in Figure 4.

If a load reads from the latest store to a variable, then the behavior is SC. If a load reads from an earlier store, then we say it reads a *stale* value. If a load reads from a store that has not yet happened, then we say it reads a *future* value. (Admittedly, concepts such as "latest" and "before" are not well defined in a concurrent execution in which operations are not ordered by happens-before. However, these concepts *are* well defined in the context of a dynamic analysis that observes conflicting operations to each variable in some global order.) HBMM permits loading both stale and future values. The failing behaviors in Figures 1 and 4 can be produced by using stale values. However, to produce failing behavior in Figure 2, future values are needed.

Furthermore, future values can sometimes produce behaviors different from those produced by stale values. For example, in Figure 5, using stale values can cause non-termination, while only future values allow the assertion to fail.

An important caveat of HBMM is that it does *not* guarantee SC for data-race-free executions—the crucial guarantee mandated by DRF0. HBMM is thus *not* strictly stronger than DRF0, rendering HBMM unsuitable as a language memory model.[2] Figure 6 shows an example of non-SC behavior allowed by HBMM but not by DRF0. This program is data race free because every SC execution only executes loads. However, under HBMM, each load can speculatively return 1, diverting the control paths to store 1 to x and y, justifying the initial speculative loads.

[2] Conversely, DRF0 allows arbitrary behavior for racy executions and is thus not strictly stronger than HBMM.

```
Initially x = y = 0

Thread 1:          Thread 2:

r = x;             while (y == 0) { }
y = 1;             x = 1;

assert r == 0
```

**Figure 5.** Using stale values, the execution may not terminate. Using future values, the assertion can fail.

```
Initially x = y = 0

Thread 1:          Thread 2:

r1 = x;            r2 = y;
if (r1 == 1)       if (r2 == 1)
  y = 1;             x = 1;

assert r1 == 0 && r2 == 0
```

**Figure 6.** An example data-race-free program that can fail its assertion under HBMM but not DRF0 [2, 11, 12].

```
Initially x = y = 0

Thread 1:          Thread 2:

r1 = x;            r2 = y;
y = r1;            x = r2;

assert r1 != 42
```

**Figure 7.** An example out-of-thin-air result [12, 34, 47].

***Java memory model.*** JMM is a strictly stronger memory model than both DRF0 and HBMM [34]. It not only enforces SC for data-race-free executions, but it tries to prohibit results that could compromise memory and type safety. (JMM introduces a concept called *causality* to define what behaviors are permitted [34].)

Figure 7 shows a canonical example [12, 34, 47] of behavior that JMM prohibits but HBMM (and DRF0) allow. HBMM permits an execution in which each load reads 42. This execution is possible as follows: each load's value is justified by a store on the other thread, which in turn is justified by the load on the same thread. To see why this behavior might conceivably happen, consider a compiler optimization that modifies each thread's code to speculatively use a predicted value (e.g., 42) at each load, and then checks the value after the store.

***Out-of-thin-air results.*** Prior work refers to behaviors such as Figure 7 as *out-of-thin-air* (OOTA) results. Prior work has not generally agreed on what constitutes an OOTA result [12, 34, 47]. In this paper, we reuse the following informal definition of OOTA results: "results that can be justified only via reasoning that is in some sense circular" [12]. Under this definition, Figure 6's non-SC behavior and Figure 7's assertion failure are OOTA results.

Figure 8 shows another OOTA example from prior work [47]. JMM only permits executions in which r2 = y sees 0. However, HBMM additionally permits executions in which r2 = y sees 1. To see why this behavior is possible, suppose that the loads of x and y see the value 1. The resulting execution (racily) stores 1 to x and y, justifying the value seen by the initial loads.

The OOTA behavior in this example actually happens in commercial JVMs [47]. A JVM's just-in-time, optimizing compiler can eliminate the redundant load of y at line 5, replacing it with r3 = 1. This transformation in turn allows x = 1 on both control paths, which in turn allows r2 to load a value of 1 from y.

```
                    Initially x = y = 0

Thread 1:                          Thread 2:

1 r1 = x;                          3 r2 = y;
2 y = r1;                          4 if (r2 == 1) {
                                   5   r3 = y;
                                   6   x = r3;
                                   7 } else x = 1;

                    assert r2 == 0
```

**Figure 8.** An example program in which compiler transformations can violate JMM [47].

We make the following observation: *in order to produce OOTA results, an execution must use future values* (e.g., the OOTA results in Figures 6, 7, and 8 rely on future values). Other unexpected and counterintuitive, yet JMM-compliant behaviors, such as the assertion-violating behavior in Figure 2, also require future values. Since real-world JVMs neither conform to the JMM nor prevent OOTA results, it is useful to expose all possible but unexpected results due to future values, whether or not they are OOTA, as long as they conform to both HBMM and DRF0.

### 2.2 Exposing Weak Memory Model Behaviors

Despite much effort, data races are widespread. By developing and evaluating analyses that expose weak memory model behaviors, researchers have demonstrated that many real data races lead to harmful behaviors. However, existing analyses have not exposed the full range of possible behaviors—particularly behaviors due to future values, which are uniquely difficult to expose.

***Data races are ubiquitous.*** Data races are hard to avoid, detect, and eliminate. Data races—and the erroneous behaviors caused by them—manifest only under certain thread interleavings, inputs, and environments, making them difficult to detect and reproduce. In spite of much research on detecting and eliminating data races (e.g., [1, 7, 13, 18–23, 37, 38, 41, 43, 44, 49–51]), there remains a fundamental tension between soundness, precision, and performance. Furthermore, programmers often introduce data races intentionally in their efforts to improve performance and scalability and avoid deadlock [9, 10, 28, 29].

***Data races are harmful.*** The conventional wisdom for decades, persisting even to the present day, is that many data races are "benign." This misconception has been solidified in part by prior work that exposes behaviors in racy executions, but considers only SC behaviors [28, 39, 45]. More recent research considers non-SC behaviors, and shows that many data races are demonstrably harmful [17, 24, 29]. However, existing dynamic analyses have been limited to exposing effects due to *stale* values. *Adversarial memory* (AM) is one such analysis, which we compare against and present in Section 4. One contribution of our work is to broaden the exploration of what kinds of harmful behaviors are possible due to data races.

***Exposing behaviors due to future values.*** Since AM and other dynamic analyses simulate behaviors due to stale values only [17, 24, 29], they cannot expose behaviors due to future values (e.g., Figures 2 and 5). An existing *model checker* called *CDSChecker* uses future values in the context of accesses to C++ atomic variables [40] (Section 9). CDSChecker explores program behaviors exhaustively but is limited to analyzing small libraries. In contrast, our work addresses the problem of how to use future values in dynamic analysis, i.e., how to use future values successfully in a single execution.

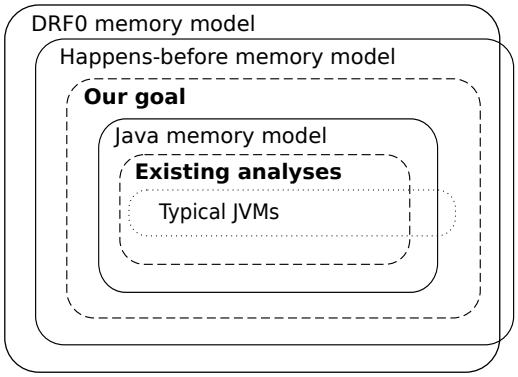



**Figure 9.** Illustration (inspired by prior work [24]) of the behaviors permitted by various memory models (solid lines), exposed by dynamic analyses (dashed lines and bold text), and exposed by typical JVMs (dotted line).

Exposing behaviors due to future values is not straightforward. *Using* a future value may affect whether the value is actually eventually stored—a necessary condition for a valid execution. Consider the example program in Figure 3 (from Section 1). An assertion failure is *not* possible under HBMM because the store to x of 1 is control-dependent on the value loaded from y. Thus, a dynamic analysis that exposes behaviors due to future values must validate that future values that are used by loads, are later stored.

This paper seeks to expose erroneous behaviors due to future values on data races, such as in Figure 2, without exposing behaviors that are *not* permitted under DRF0 or HBMM, such as the assertion failures in Figures 3 and 6. Notably, we argue that exposing OOTA behaviors such as in Figures 7 and 8 is worthwhile since (1) JVMs actually allow some OOTA behaviors, and (2) it is still unclear what behaviors JMM should allow or forbid.

Figure 9 illustrates the behaviors allowed by several memory models, compared with behaviors exposed by prior dynamic analyses and targeted by this paper's analysis. Neither DRF0 nor HBMM is a subset of the other, as Section 2.1 explained. JMM permits behaviors that are a strict subsect of the intersection of DRF0 and HBMM. Typical JVMs do not conform to JMM, and existing analyses for Java programs can only expose a subset of behaviors allowed by JMM. Our goal is to expose not only behaviors allowed by the JMM, but also behaviors allowed by the intersection of DRF0 and HBMM.

## 3. Preliminaries: Common Notation

This section introduces notation that we use to present both prior work's adversarial memory (AM) analysis [24] in Section 4 and our prescient memory (PM) analysis in Sections 5 and 6.

We use the following notation to describe a multithreaded execution:

$\boldsymbol{t}$ : A thread identifier.

$\boldsymbol{x}$ : A shared-memory variable.

$\boldsymbol{v}$ : A value loaded from or stored to a variable.

The analyses are mainly concerned with memory access operations, which are each one of the following:

$\boldsymbol{rd(t, x)}$ : Thread $t$ loads from variable $x$.

$\boldsymbol{wr(t, x, v)}$ : Thread $t$ stores a value $v$ to variable $x$.

Both the AM and PM analyses rely on the classic *vector clock algorithm* [23, 35] to track logical time and happens-before rela-

**Algorithm 1** STORE [AM]: $wr(t, x, v)$

$W_x \leftarrow W_x \cdot v@C_t$

**Algorithm 2** LOAD [AM]: $rd(t, x)$

**let** $v \leftarrow pick(visible(W_x))$
**return** $v$

tions [30]. The algorithm associates a vector clock with each thread and each synchronization object (in Java, every object and volatile field), and updates these vector clocks at synchronization operations (lock acquire and release, monitor wait, thread fork and join, and volatile accesses). The vector clock algorithm uses the following notations:

$\boldsymbol{K}$ : A vector clock, which maps each thread to an integer [35].

$\boldsymbol{K \sqsubseteq K'}$ : This relationship means that for every thread identifier, $K$'s integer is less than or equal to $K'$'s integer. The vector clock algorithm ensures that if an event at logical time $K$ *happens before* an event at time $K'$, then $K \sqsubseteq K'$. Otherwise $K \not\sqsubseteq K'$.

$\boldsymbol{C_t}$ : Represents thread $t$'s current vector clock.

## 4. Prior Work: Adversarial Memory

Prior work introduces *adversarial memory* (AM) [24] to expose a subset of behaviors that are allowed under the Java memory model (Section 2.1). AM enables loads to see *stale* values by buffering an execution's stores and tracking happens-before relations. Like other existing dynamic analyses [17, 29], AM does *not* allow loads to see *future* values.

AM associates a *write buffer* with every shared variable:

$\boldsymbol{W_x}$ : A write buffer for variable $x$, which has the following form:

$$W_x = v_1@K_1 \cdot v_2@K_2 \cdot \ldots \cdot v_n@K_n$$

where $n > 0$. Each pair $v@K$ denotes that a store of value $v$ to $x$ was executed at vector clock timestamp $K$. The write buffer initially contains only $0@\bot$ (where $\bot$ is the vector clock that maps all threads to 0), representing that the variable is initialized to its default value [33].

Algorithms 1 and 2 show the analysis that AM performs at each program store and load, respectively. At each store, AM appends the current $v@C_t$ to the write buffer. At each load, AM picks a value from the *visible set* of values from the write buffer. AM picks a value using a heuristic function $pick()$ (definition not shown). Our experiments reuse heuristics from prior work, which include picking the oldest value, picking the oldest value that is different from the last returned value, and picking a random value from the visible set [24].

Algorithm 3 defines the function $visible()$ for computing the visible set. The condition in braces specifies what values in the write buffer are legal for the current load to see. Since every value in the write buffer is the result of a concrete, previously executed store, a load cannot see values from future stores. Thus, the rule only needs to ensure that there is no intervening store. A value $v_i$ is legal to return if there is no subsequent store of $v_j$ ($i < j \leq n$) that (1) happens after the store of $v_i$ and (2) happens before the current load. The returned visible set maintains the same order of the values as they appear in $W_x$.

**Algorithm 3** Helper function

**function** $visible(W_x)$
  **return** $\{v_i \mid 1 \leq i \leq n \ \wedge (\nexists j > i \ . \ K_i \sqsubseteq K_j \sqsubseteq C_t)\}$

**Algorithm 4** LOAD [PM]: $rd(t, x)$

1: **let** $v \leftarrow latest(W_x)$ ▷ Start with current value
2: $v \leftarrow predict(v, \ldots)$
3: **if** $v \notin visible(W_x)$ **then** ▷ Is $v$ a speculative value?
4: $\quad S_x \leftarrow S_x \cup \{\langle C_t, v\rangle\}$
5: **return** $v$

**Algorithm 5** STORE [PM]: $wr(t, x, v)$

1: $W_x \leftarrow W_x \cdot v@C_t$
2: **for all** $\langle K, v'\rangle \in S_x$ **do**
3: $\quad$ **if** $K \not\sqsubseteq C_t \wedge v' = v$ **then** ▷ Is speculative load validated?
4: $\quad\quad S_x \leftarrow S_x - \{\langle K, v'\rangle\}$

## 5. Prescient Memory

A key limitation of AM and related existing work [17, 24, 24, 28, 29, 39, 45] is the inability to "look into the future" and load a *future value* (Section 2). This limitation means that these analyses cannot expose some behaviors that are allowed under weak memory models. To overcome this limitation, we introduce an analysis called *prescient memory* (PM), which supports using and validating future values. The behaviors exposed and validated by PM are allowed under the happens-before memory model (HBMM; Section 2.1). However, PM *as presented in this section* can expose *non-DRF0* behaviors, by producing non-SC results in data-race-free programs such as Figure 6 (from Section 2). In contrast, Section 6 introduces a *PM workflow* that exposes non-SC behaviors only for programs with data races.

Since every legitimately loaded future value is the result of a future store, PM performs *speculative* loads to "guess" a future value. At later stores to the same variable, PM tries to *validate* each speculative load by checking if any concurrent store (i.e., a store that races with the load) actually stores the same value that the load used.

PM uses the same notation as AM, and it calls the $visible()$ function from Algorithm 3. PM maintains the same state as AM (the write buffer $W_x$) and the following additional state:

$\boldsymbol{S_x}$ : A *speculative read history* for variable $x$. $S_x$ contains tuples of the form $\langle K, v\rangle$, which denotes that a load of $x$ at time $K$ used a speculative value $v$. Initially $S_x$ is $\emptyset$.

Algorithm 4 shows the analysis that PM performs at a load. The analysis picks a value that is predicted to be a future value, using the $predict()$ function (line 2), which may elect to return the provided *latest* value instead of a potential future value. (The ". . ." in $predict()$'s parameter list represents additional parameter(s) that could be provided to help prediction. Section 6 introduces an additional parameter that encodes dynamic program location.) If the predicted value is not in the visible set, then the load operation is speculative, and the analysis records the load in $S_x$ (line 4).

Algorithm 5 shows the analysis at a program store. In addition to appending the current $v@C_t$ to $W_x$ (line 1), the algorithm checks if the current store validates a previously executed speculative load from $S_x$ (line 3). The algorithm removes all such matching entries from $S_x$ (line 4).

*An execution is valid only if (and when) all speculative loads have been validated.* Algorithm 6 checks at program termination

**Algorithm 6** TERMINATION [PM]

```
1: for all x do
2:     if S_x ≠ ∅ then
3:         Invalid execution!
```

**Algorithm 7** EARLY VALIDATION [PM]: $x$

```
1: for all ⟨K, v⟩ ∈ S_x do
2:     if ∀t, K ⊑ C_t then          ▷ Can v never be validated?
3:         Invalid execution!
```

if all speculative loads have been validated. If not, the current execution may not conform to HBMM, and any erroneous behavior it exhibits is not worth investigating.

An execution may fail to terminate normally, by throwing an exception or getting stuck in an infinite loop, as a result of a speculative load. This behavior should be considered to be legal under HBMM if and only if PM can validate all speculative loads (i.e., $\forall x \,.\, S_x = \emptyset$). Otherwise, the behavior is invalid: it is quite possibly not allowed under HBMM.

It is sometimes possible to determine prior to program termination that an unvalidated speculative load can never be validated. Algorithm 7 shows the logic, which can be invoked at any point during program execution. If a speculative load *happened before* all threads' current vector clocks (line 2), any future stores will *happen after* this speculative load—so the load will *never* be validated. In a managed language such as Java, it is convenient to implement Algorithm 7 at garbage collection (GC) time, since (full-heap) GC traverses all shared variables $x$, at which point it can process $S_x$.

## 6. Making Prescient Memory Practical

PM as described in Section 5 presents two main challenges:

1. It is difficult to make PM efficiently produce a valid execution containing future values for large, real programs. In particular, how should PM choose when and which future values to use, what actions can it take to improve the chances that future values will be validated? Prior work that has used and validated future values has not dealt with this challenge; instead it performs model checking of small programs [40] (Section 9).
2. PM can expose behaviors that are possible under HBMM but not DRF0. As a result, PM can expose non-SC behaviors even for data-race-free programs, such as the assertion failure in Figure 6.

We address the above challenges using a novel approach called the *PM workflow* (or simply the *workflow*) that consists of the following components:

- profiling of potential future values;
- predicting which loads should use future values, and which values to use; and
- deterministic replay that helps provide consistent behavior between profiling and predicting runs, while also permitting divergence as needed.

Figure 10 illustrates the workflow. The workflow limits analysis to memory accesses involved in data races, identified in separate program execution(s) using a dynamic data race detector, as in prior work that uses stale values [17, 24, 29]. By limiting PM only to accesses involved in data races, the workflow avoids producing non-SC results for data-race-free programs.

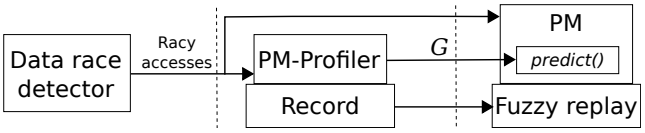



**Figure 10.** Overview of the PM workflow. Dashed lines separate distinct program executions.

**Algorithm 8** STORE [PM-profiler]: $wr(t, x, v)$

```
1: W_x ← W_x · v@C_t
2: for all ⟨l, K, {v_1, v_2, ...}⟩ ∈ R_x do
3:     if K ⋤ C_t ∧ v ∉ {v_1, v_2, ...} then   ▷ Potential future value?
4:         G_l ← G_l ∪ {v}
```

**Algorithm 9** LOAD [PM-profiler]: $rd(t, x, l)$

```
1: R_x ← R_x ∪ {⟨l, C_t, visible(W_x)⟩}
2: let v ← latest(W_x)
3: return v
```

The rest of this section describes specific challenges and how the workflow's components address them.

### 6.1 Profiling Potential Future Values

In order to assist PM's prediction of future values, we introduce a separate analysis called *PM-profiler* that produces a set of promising future values for each executed load.

PM-profiler and PM need a mechanism to identify executed load operations. We introduce the following identifier:

$\boldsymbol{l}$ : A *dynamic program location* that uniquely identifies a dynamic load. $l$ encodes both the thread that executed the load and the dynamic instance of the static instruction.

PM-profiler maintains the following data structures:

$\boldsymbol{R_x}$ : A *concrete read history* for variable $x$. Each element in $R_x$ is a tuple with the form $\langle l, K, \{v_1, v_2, \ldots\}\rangle$. The set $\{v_1, v_2, \ldots\}$ is a non-empty set of all visible values at the load operation. Initially $R_x$ is $\emptyset$.

$\boldsymbol{G_l}$ : The "promising" future value set for a load operation identified by $l$. $G$ is the interface between PM-profiler and PM: PM-profiler produces $G$, and PM's $predict()$ function uses $G$ as a read-only dictionary.

PM-profiler's analysis at a program store, shown in Algorithm 8, identifies promising future values. It checks if the current store can provide a future value for any of the previous loads in $R_x$ (lines 2–4). If the current store is concurrent with a previous load, and the store provides a value that is distinct from any value in the visible set of the load (line 3), then the analysis records the value of the store in the set of promising future values for the load (line 4).

At a program load, PM-profiler computes and stores the visible set for the load, as Algorithm 9 shows. The analysis records the dynamic program location $l$, the current time $C_t$, and the visible set in $R_x$. It always returns the latest value from $W_x$; PM-profiler does *not* try to expose any weak memory model behaviors.

### 6.2 Predicting Future Values

We now overview the prediction component of the workflow, which is represented with the call to function $predict()$ in Algorithm 4 (from Section 5). In order to use $G$ (the potential future values

produced by PM-profiler), PM passes $l$ to $predict()$; that is, PM's analysis in Algorithm 4 calls $predict(v, l)$.

There are two different questions for prediction: which loads should use future values, and which future values should they use? We find that in practice only the first question matters: most loads with future values have only one future value, and using different future values for loads with multiple future values has little impact on the behaviors that PM can expose.

Thus, $predict()$ is concerned with choosing which loads with a future value (i.e., loads at $l$ with non-empty $G_l$) should use a future value. (When there are multiple values in $G_l$, $predict()$ chooses one randomly.) In general, if more loads return future values, the execution is more likely to exhibit new, potentially erroneous behaviors. On the other hand, using more future values means that the execution is less likely to be able to validate every loaded future value.

Our implementation of $predict()$ supports the following policies:

**All** : Every load with non-empty $G_l$ uses a future value. Except for microbenchmarks, this policy almost always leads to validation failures. However, it is useful for exposing behaviors that *might* be possible if the "right" set of loads were selected to use future values.

**Selective** : The first $k$ executed loads do *not* use future values, and the following $m$ executed loads use future values. Skipping $k$ executed loads helps when we have identified that some loads either (1) have future values that PM cannot validate successfully or (2) have future values that can be validated and lead to erroneous behavior, but we want to look for other behaviors later in the execution. Using future values for the next $m$ loads only, increases the chances that all future values will be validated.

**Per-site** : This policy modifies the previous policy so that prediction applies only to one particular static load site (i.e., static program location that performs a load). By running with this policy separately for each site, we can increase the chances of finding a load that produces a future value that can be validated.

Our evaluation tries various combinations of these in order to find future values that can be validated and to expose erroneous behaviors. We also tried introducing randomness into the above policies, but did not uncover any new behaviors as a result.

***Behaviors exposed by the PM workflow.*** The PM workflow exposes not only behaviors allowed by JMM but also additional behaviors permitted by both HBMM and DRF0, i.e., the behaviors labeled "Our goal" in Figure 9 (from Section 2). Figure 8 (from Section 2) shows an example of such behavior. While some of these behaviors likely cannot be exposed by any conceivable JVM optimization, these behaviors are still of interest to developers, particularly since the exact set of possible behaviors that JVM optimizations might allow is ill defined.

The PM workflow allows some strange behaviors that are still DRF0—depending on one's exact definition of DRF0. Consider Figure 11, for which the PM workflow can cause the assertion to fail. PM-profiler identifies the racy store `x = 1` by Thread 1. However, when PM uses that value at Thread 2's load, the data race on `z` would *not* manifest and Thread 1's store to `x` does *not* execute. Instead, Thread 3's store to `x` validates the load.

### 6.3 Fuzzy Deterministic Replay

Multithreaded executions are inherently nondeterministic due to timing-sensitive thread interleavings. This nondeterminism presents two challenges for PM-profiler and PM, which operate on separate executions. First, nondeterminism makes it difficult to match loads across program executions: the mapping $G$ produced by PM-profiler is unlikely to be useful to PM if program execution diverges. Second, nondeterminism makes it less likely that a potential future value from a prior execution will actually be validated by a future store.

```
Initially x = y = z = 0

Thread 1:          Thread 2:          Thread 3:

                   r2 = x;            r3 = y;
                   if (r2 == 1)       if (r3 == 1)
                     y = 1;             x = 1;
                   else
                     z = 1;
r1 = z;
if (r1 == 1)
  x = 1;

                   assert r3 == 0
```

**Figure 11.** An example program for which the PM workflow can cause the assertion to fail.

Our workflow thus extends *multithreaded record & replay* [14, 32, 48] in order to eliminate nondeterminism between the PM-profiler and PM executions. Deterministic replay helps guide the PM execution to match the PM-profiler execution's thread interleavings. However, *after* PM uses a future value, execution may *diverge* from the recorded execution. Nonetheless, we have found that deterministic replay is still useful at this point in order to potentially guide the execution to store (and thus validate) the future value.

In some cases, divergence could cause the deterministic replay mechanism to be unable to continue. For example, suppose thread `T2` is waiting for thread `T1` to reach a specific execution point in order to ensure deterministic replay. If `T1` loads a future value and diverges from the recorded execution, `T1` may never reach the point that `T2` is waiting for, in which case replay is "stuck." Instead of failing the execution, PM detects when replay is stuck and proceeds *without being guided by replay*. We refer to this best-effort replay approach as *fuzzy replay*. Fuzzy replay is useful not only for validating future values at upcoming stores, but also for *using* additional future values at upcoming loads after the execution has already diverged.

## 7. Implementation

We have implemented AM, PM, and the PM workflow in Jikes RVM 3.1.3 [5, 6], a high-performance Java virtual machine (JVM) that performs competitively with commercial JVMs [7]. Our implementation of the PM workflow builds on existing, publicly available implementations of dynamic data race detection (the FastTrack algorithm [23] implemented in Jikes RVM [7]) and multithreaded record & replay [14]. Our implementation of AM is influenced by a publicly available implementation of AM in Jikes RVM [46].

AM, PM, and PM-profiler modify Jikes RVM's dynamic compilers to add instrumentation at every memory access identified by the data race detector. The analyses bound the size of each variable's write buffer ($W_x$) and read history ($R_x$ for PM-profiler; $S_x$ for PM) in order to avoid running out of memory. The implementations represent dynamic program location $l$ as a tuple of the thread, the static site (method and bytecode index), and a per-thread, per-site counter.

We extend the existing record & replay implementation to support our workflow. We extend the record and replay analyses to record and replay synchronization operations (which normally would be ignored [14]), so PM can perform the vector clock al-

| Program | Field(s) involved | Worst erroneous behavior | | | |
|---|---|---|---|---|---|
| | | AM | (Observable?) | PM workflow | (Observable?) |
| Figure 1 | x ,y | Assertion failure | (Yes) | None | (N/A) |
| Figure 2 | x, y | None | (N/A) | Assertion failure | (Yes) |
| Figure 3 | x, y | None | (N/A) | None | (N/A) |
| Figure 4 | x | Divide-by-zero | (Yes) | Divide-by-zero | (Yes) |
| Figure 5 | x, y | Non-termination | (Yes) | Assertion failure | (Yes) |
| Figure 6 | None* | None | (N/A) | None | (N/A) |
| Figure 7 | x, y | None | (N/A) | None | (N/A) |
| Figure 8 | x, y | None | (N/A) | Assertion failure | (Yes) |
| Figure 11 | x, z | None | (N/A) | Assertion failure | (Yes) |
| hsqldb6 | MemoryWatcherThread.keep_running | Non-termination | (Yes) | Data corruption | (Yes) |
| hsqldb6 | JavaSystem.memoryRecords | None | (N/A) | Performance bug | (No) |
| avrora9 | Transmission.lastBit | Data corruption | (Yes) | Data corruption | (Yes) |
| lusearch9 | ThreadLocal.nextHashBase | Performance bug | (No) | None | (N/A) |
| sunflow9 | Geometry.builtAccel | Null ptr exception | (Yes) | Null ptr exception | (Yes) |
| pjbb2000 | Company.mode | Non-termination | (Yes) | Data corruption | (Yes) |
| pjbb2000 | Company.elapsed_time | Data corruption | (Yes) | Data corruption | (Yes) |
| pjbb2005 | DomNode.eventDataLock | Data corruption | (No) | Data corruption | (No) |
| pjbb2005 | DomEvent.stop | Data corruption | (No) | None | (N/A) |

**Table 1.** Summary of erroneous program behaviors discovered by returning stale or future values. *The program in Figure 6 is data race free, so AM and the PM workflow do not instrument any memory accesses.

gorithm. To support fuzzy replay, we extend replay so that it stops trying to replay if it gets stuck or encounters a replay error. In order to support multiple replay attempts from one recorded execution, we extend the "fork-and-restart" mechanism that the record & replay implementation uses [14].

A limitation of our current implementation is that it can use future values with primitives types but not reference types (i.e., references to objects). This limitation exists because object *addresses* are not in general the same across record and replay, since the record & replay implementation provides *application-level determinism* [14]. That said, it should be possible to use reference types by extending our implementation. Objects can be identified uniquely across runs by tagging each object with its thread and a per-thread counter incremented at each allocation (the record & replay implementation already identifies objects this way in order to provide deterministic hash codes [14]), although this approach would seem to require the ability to look up any object based on its tag during replay. Another challenge is that at a load to a reference-type future value, the object may not have been *allocated* yet. The implementation could address this by eagerly allocating (but not initializing) the object at the load.

## 8. Evaluation

This section evaluates the PM workflow's ability to expose erroneous program behaviors using future values. In this section, "PM" refers to the PM analysis executing as part of the PM workflow (Section 6), *not* the general form of PM from Section 5.

### 8.1 Methodology

Our experiments execute benchmarked versions of real applications: the DaCapo benchmarks, versions 2006-10-MR2 and 9.12-bach (2009) [8] (limited to multithreaded programs that Jikes RVM can run), and fixed-workload versions of SPECjbb2000 and 2005.[3]

We build a high-performance configuration of Jikes RVM. The experiments run on a system with an Intel Core i5-2500 4-core processor running Linux 2.6.32. (We also tried running experiments on a 32-core machine, but that did not expose any new behaviors.)

[3] `http://spec.org/`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

For a fair comparison between AM and PM, the experiments only consider fields of non-reference types, since the PM implementation does not currently support reference types (Section 7). We execute AM and PM repeatedly for each program, trying out different AM heuristics [24] and PM prediction policies (Section 6.2) to see what kinds of erroneous behaviors can be exposed, such as corrupted output, exceptions, and non-termination.

### 8.2 Exposing Erroneous Behavior

Table 1 summarizes erroneous behaviors discovered by our implementations of AM and PM. For completeness, our evaluation includes results for the example programs in Figures 1–8 and Figure 11. For the 12 real programs we evaluated, PM exposes 7 erroneous behaviors. Of these 7 bugs, AM can expose the same behaviors for 4 of them, AM exposes different behaviors for 2, and AM cannot expose the bug for 1. Additionally, AM can expose erroneous behavior for 2 bugs for which PM cannot expose erroneous behavior. In some cases, the same error manifests differently in AM and PM, e.g., non-termination versus data corruption.

Interestingly, PM exposes erroneous behavior for most of the same bugs for which AM exposes erroneous behavior, even though PM does *not* use stale values. Our evaluation intentionally compares analyses with non-overlapping functionality: AM uses only stale values, while PM uses only future values. A more powerful analysis would ideally combine AM and PM in order to load both stale and future values.

In the evaluated real programs, PM does not expose any out-of-thin-air (OOTA) results. Nonetheless, researchers and practitioners could use the PM workflow to identify OOTA behaviors, including controversial and/or JMM-violating behaviors. Any real-world evidence of such behaviors would inform future revisions to language specifications.

Our experiments detect a few stale and future values beyond those reported in Table 1. For 5 fields (2 in hsqldb6, 2 in avrora9, and 1 in sunflow9), AM detects stale values but cannot expose erroneous behavior. For the field in sunflow9, PM-profiler detects future values, which PM can use and validate, but it cannot expose erroneous behavior. Other than these cases and the cases in Table 1, there are no fields for which AM detects stale values or PM-profiler

detects future values (including future values that PM cannot validate).

***Microbenchmarks.*** The table shows that PM and AM behave as expected for the microbenchmarks corresponding to Figures 1–8 and Figure 11. Although the general form of PM presented in Section 5 can expose erroneous behavior for Figures 6 and 7, the PM workflow cannot.

***hsqldb6.*** This database management system has a thread that continuously monitors the application's memory usage and uses a boolean flag MemoryWatcherThread.keep_running as a termination condition. Threads access this variable racily. Returning a stale value can prevent the thread (and consequently the whole program) from terminating. Returning a future value can cause the thread to terminate early and corrupt memory usage statistics. By default, the benchmarked version of the program does not output the statistics, but we have modified it to do so, making the data corruption visible.

JavaSystem.memoryRecords is a counter that the program increments at certain memory operations. The program periodically checks the counter to decide if it should trigger garbage collection (GC):

```
if (memoryRecords > n) { // n is a run−time constant
  memoryRecords = 0;
  System.gc();
}
```

Returning a stale value can trigger GC less frequently, but triggering GC is unnecessary since the JVM does it automatically. (Furthermore, JVMs are permitted to ignore System.gc() calls [33].) Returning a future value can trigger GC more frequently; PM is able to successfully use and validate future values. In theory, repeatedly using a large future value could cause a performance bug by triggering GC frequently. However, we have been unable to produce PM executions that use future values that can be validated but are also large enough to cause noticeable slowdowns.

***avrora9.*** This program is a simulator for an embedded microcontroller. Transmission.lastBit is a long field that indicates the end-byte position of a simulated radio transmission. The program uses this field to compute a list of intersecting transmissions and other simulation metrics:

```
1  List getIntersection (long bit) {
2    List it = null;
3    synchronized (medium) {
4      Iterator i = medium.transmissions.iterator();
5      while (i.hasNext()) {
6        Transmission t = (Transmission) i.next();
7        if (it == null) it = new LinkedList();
8        if (bit >= t.firstBit && bit < t.lastBit) {
9          it.add(t);
10       }
11     }
12   }
13   return it;
14 }
```

Loading from this field (line 8) can return stale values and future values that can be validated. In both cases, the values lead to an incorrect list and corrupt the resulting metrics. However, this corruption is infrequent in our experiments. In 500 trials each for AM and PM, we found that AM and PM corrupted output in 23 trials (4.6%) and 10 trials (2.0%), respectively. It is easier to expose this bug using AM: all AM executions are always legal under HBMM (and JMM, in fact). For PM, more than half of the 500 executions failed to validate every future value, making them invalid even though they corrupted output in some cases.

***lusearch9.*** This program uses the lucene indexing and search library to perform text search. The program uses a field ThreadLocal.nextHashBase, which is part of GNU Classpath, the Java library implementation used by Jikes RVM. The library uses the counter to initialize a final field, hashCode, for each new ThreadLocal instance. The method that increments nextHashBase is synchronized but (erroneously) not static. We note that this bug should be attributed to GNU Classpath, not the lusearch9 benchmark or lucene library.

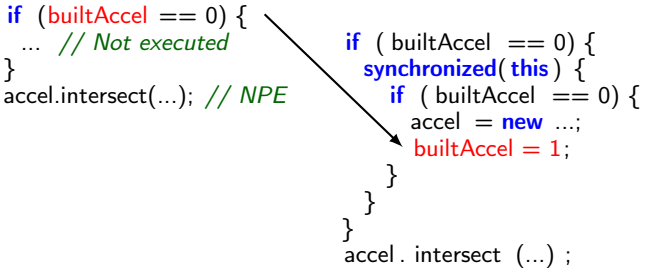
Two object instances can share the same hash code value, as long as the hashCode field remains constant after initialization, which could lead to a performance bug by increasing the chances of hashing collisions. However, ThreadLocal is used infrequently in this program, so a performance bug is not observable. Future values could in theory lead to a performance bug by using the same future value for many loads, but our experiments cannot successfully validate executions in which many loads use future values.

***sunflow9.*** The program uses a double-checked locking pattern to lazily initialize the shared reference Geometry.accel (code simplified from the original):

```
1 if (builtAccel == 0) {
2   synchronized(this) {
3     if (builtAccel == 0) {
4       accel = new ...;
5       builtAccel = 1;
6     }
7   }
8 }
9 accel.intersect(...);
```

The accesses to accel and int field builtAccel are racy because the program fails to declare builtAccel as volatile. Returning a future value of 1 for builtAccel at line 1, can trigger a null pointer exception (NPE) at line 9. (AM is also able to expose this bug if it instruments accesses to the reference-type field accel, which can return a stale value of null at line 9.)

The following figure shows an interleaving that PM can produce by using future values. The arrow connects the racy accesses that load and later store the future value 1.




Accesses to another int field Geometry.builtTess use a similar racy double-checked locking pattern. However, returning a stale value or a future value on this field does not lead to any erroneous behavior that we could detect.

***pjbb2000.*** This program is an artificial benchmark that simulates the backend of a business server. It uses a field Company.mode to maintain the state of a Company object. The program uses the following unsynchronized load of mode to decide whether to update statistics data:

```
1  static boolean eventDataLock = false;
2  static Object lock = new Object();
3  static DomMutationEvent m = new
       DomMutationEvent();

4  void insertionEvent(DomNode target) {
5    boolean doFree = false;
6    DomMutationEvent e = null;
7    synchronized(lock) {
8      if (!eventDataLock) {
9        eventDataLock = true;
10       doFree = true;
11       e = m;
12     }
13   }
14   if (e == null) {
15     e = new DomMutationEvent();
16   }
17   e.initialize(...);
18   target.dispatchEvent(e);
19   if (doFree) {
20     e.clear();
21     eventDataLock = false;
22   }
23 }
```

**Figure 12.** Code from pjbb2005.

```
if (company.mode == Company.RECORDING)
  myTimerData.updateTimerData(txntype, txntime);
```

PM returns a future value of Company.RECORDING at this load, leading the program to take the true branch, which should not be taken until later in the execution, corrupting reported statistics.

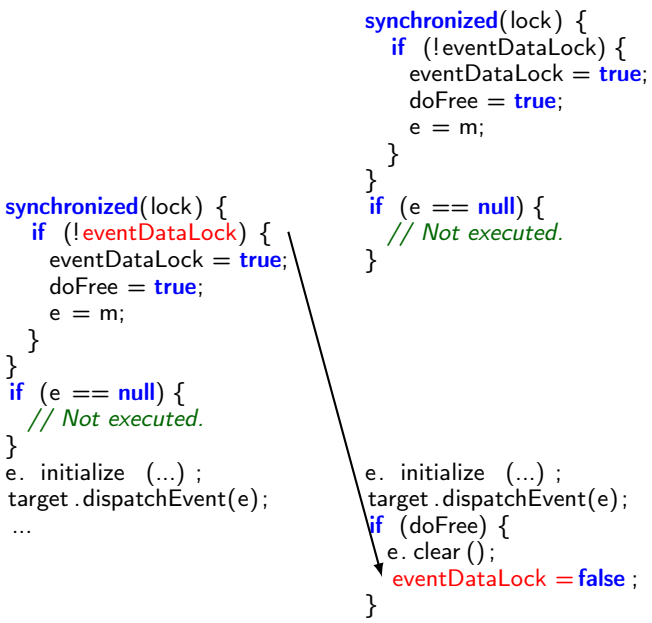
Returning a stale value cannot trigger this data corruption, because the value Company.RECORDING does not exist in the set of stale values. Nonetheless, using a stale value for a *different* program load of mode can lead to non-termination. For this other load, PM-profiler detects no future value.

For another field Company.elapsed_time, both stale and future values lead the program to report an incorrect timing value, corrupting the output statistics. (Table 1 thus reports the erroneous behavior as "observable." However, the behavior may be hard to observe in practice because the program is an artificial benchmark targeting performance testing and does not have a clear specification for correct output, which is nondeterministic from run to run.)

***pjbb2005.*** This artificial benchmark invokes the following code in XML processing libraries that are part of the GNU Classpath implementation. DomNode.eventDataLock is a static boolean field that helps to enforce mutual exclusion. The program minimizes allocations of DomMutationEvent objects using code shown in Figure 12.

Consider the following scenario. One thread initializes a DomMutationEvent object (line 17) and dispatches it to a target node object (line 18). Instead of allocating a new DomMutationEvent object every time, the code tries to reuse the shared "scratch" object referenced by m (lines 7–13). The eventDataLock field indicates if m is currently used by a thread. In a sequentially consistent (SC) execution, lines 17–18 execute atomically when threads reuse the shared object m.

However, "releasing" eventDataLock on line 21 is racy. This permits the load of the field on line 8 to return false even if the SC value would be true. As a result, two threads can simultaneously use the shared object on lines 17–18, violating mutual exclusion. Using either a stale value or a future value can trigger this behavior. The following illustration shows an interleaving with a future value, with the arrow connecting the load and store that use and store the future value of false, respectively.




The dispatchEvent() method (called from line 18 in Figure 12) accesses another field, DomEvent.stop. As a result of the racy "release" of eventDataLock, loads to DomEvent.stop can return a stale value, prematurely ending a traversal of a DomNode array and possibly corrupting data. PM-profiler does not detect any future values for this field.

For both fields, we have been unable to detect any visible effect from the output of pjbb2005, even though using a stale or future value can violate mutual exclusion and corrupt memory states. We suspect that since this benchmark is designed for performance testing alone, it lacks sensitivity to this data corruption. In any case, these bugs (like the lusearch9 bug) should be attributed to GNU Classhpath, not to pjbb2005.

### 8.3 Run-Time Performance

This section measures the run-time overhead added by PM, compared with AM. We run configurations of AM and the PM workflow that do *not* use any stale or future values; PM-profiler still records potential future values, and PM simulates the cost of using future values by recording them in the read history $S_x$.

Figure 13 shows the run-time overhead that each analysis adds over execution on the unmodified JVM. The average overhead of AM is 76%. PM-profiler incurs almost 600% overhead on average, while PM incurs 390%. We find that less than one-third of PM-profiler and PM's overhead comes from the record and replay analyses, respectively (results not shown).

PM-profiler and PM perform significantly more work than AM and thus add substantially more overhead. PM-profiler adds more overhead than PM since only PM-profiler tracks the concrete read history $R_x$ for each variable $x$. AM and PM add overhead proportional to the frequency of instrumented (racy) accesses, so measured overhead varies significantly across the evaluated programs. We have *not* endeavored to optimize the implementations, which for convenience use inefficient patterns (e.g., heavy use of containers with boxed primitives).

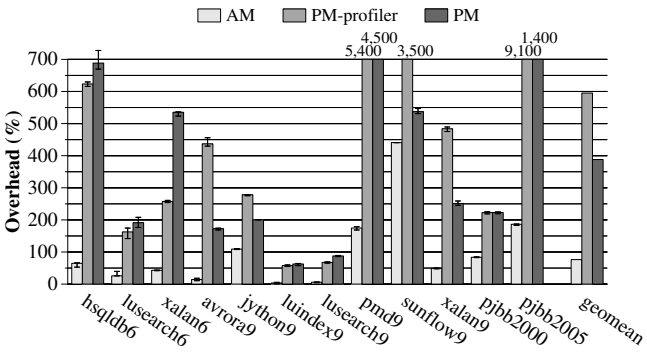



**Figure 13.** Run-time overhead of AM, PM-profiler, and PM. Each bar is the median of 10 trials. The intervals are 95% confidence intervals centered at the mean. Overheads exceeding 700% are labeled using two significant figures.

## 9. Related Work

Section 2.2 compared our prescient memory (PM) work with existing dynamic analyses that expose erroneous behavior due to data races [17, 24, 28, 29, 39, 45]. More recent analyses have generally exposed more behaviors (e.g., uses of stale values), but none of these analyses can use future values. Our work thus expands the scope of this area, providing insights into what new behaviors are legal under weak memory models.

Prior work employs model checking and verification techniques to explore concurrent program behaviors, including effects due to data races under weak memory models [15, 16, 25, 26, 36, 40]. These techniques typically offer better theoretical guarantees and greater coverage than dynamic analyses. However, model checking typically suffers from state-space explosion for realistically sized programs. Thus, these tools commonly target small portions of code such as concurrent data structures and core algorithms, instead of entire large applications.

*CDSChecker* is a model checker that exhaustively explores program behaviors allowed by the C/C++ memory model [40]. It only considers C/C++ atomic variable accesses, since racy accesses on ordinary C/C++ variables have undefined semantics. CDSChecker supports returning both stale and future values for atomic variable loads, which are permitted by the C/C++ memory model [11] with constraints similar to those in the happens-before memory model (HBMM; Section 2.1). It would be infeasible to extend CDSChecker's exhaustive approach to large, real programs, particularly for a safe language such as Java, in which every potentially racy access can have weak memory model behavior. CDSChecker's evaluation does not report any new behaviors from future values (i.e., no behaviors not already possible by using stale values).

In contrast, our work targets full applications in which any racy access has weak but defined semantics [12, 30, 34, 47]. Model checking would be impractical for these long executions with frequent relevant operations. We introduce a workflow that often enables an execution to use validatable future values. As a result, PM successfully exposes new behaviors due to future values in large applications.

## 10. Conclusion

Prescient memory (PM) enables programs to return future values at load operations via a speculation-and-validation approach. We introduce a novel, practical workflow that incorporates profiling, prediction, and fuzzy replay to help PM use future values successfully in large, real applications. Our evaluation demonstrates that this approach effectively exposes previously unknown erroneous behaviors due to future values. Thus, our work overcomes a key limitation of existing dynamic analyses that are unable to use future values, advancing the state of the art in practically exposing behaviors possible under weak memory models.

## Acknowledgments

We thank our shepherd, Doug Lea, and the anonymous reviewers for valuable suggestions for improving the paper, particularly for help with the treatment of behaviors allowed by HBMM but not DRF0. We thank Hans Boehm, Brian Demsky, and Harry Xu for helpful discussions.

## References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *TOPLAS*, 28(2):207–255, 2006.

[2] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.

[3] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *ISCA*, pages 2–14, 1990.

[4] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *ISCA*, pages 234–243, 1991.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[6] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[7] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[9] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.

[10] H.-J. Boehm. Position paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *RACES*, pages 9–14, 2012.

[11] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[12] H.-J. Boehm and B. Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*, pages 7:1–7:6, 2014.

[13] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional Detection of Data Races. In *PLDI*, pages 255–268, 2010.

[14] M. D. Bond, M. Kulkarni, M. Cao, M. Fathi Salmi, and J. Huang. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *PPPJ*, pages 90–101, 2015.

[15] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *PLDI*, pages 12–21, 2007.

[16] S. Burckhardt and M. Musuvathi. Effective Program Verification for Relaxed Memory Models. In *CAV*, pages 107–120, 2008.

[17] J. Burnim, K. Sen, and C. Stergiou. Testing Concurrent Programs on Relaxed Memory Models. In *ISSTA*, pages 122–132, 2011.

[18] M. Christiaens and K. D. Bosschere. Accordion Clocks: Logical Clocks for Data Race Detection. In *Euro-Par*, pages 494–503, 2001.

[19] M. Christiaens and K. De Bosschere. TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs. In *Symposium on Java Virtual Machine Research and Technology Symposium*, pages 15–15, 2001.

[20] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, 2007.

[21] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, pages 237–252, 2003.

[22] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, pages 1–16, 2010.

[23] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[24] C. Flanagan and S. N. Freund. Adversarial Memory for Detecting Destructive Races. In *PLDI*, pages 244–254, 2010.

[25] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[26] T. Q. Huynh and A. Roychoudhury. Memory Model Sensitive Bytecode Verification. *Formal Methods in System Design*, 31(3):281–305, 2007.

[27] A. Jeffrey and J. Riely. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. In *LICS*, 2016.

[28] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, pages 185–198, 2012.

[29] B. Kasikci, C. Zamfir, and G. Candea. Automated Classification of Data Races Under Both Strong and Weak Memory Models. *TOPLAS*, 37(3):8:1–8:44, May 2015.

[30] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[31] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Computer*, 28:690–691, 1979.

[32] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*, pages 463–474, 2012.

[33] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2nd edition, 1999.

[34] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[35] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[36] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

[37] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *POPL*, pages 327–338, 2007.

[38] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, pages 308–319, 2006.

[39] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, pages 22–31, 2007.

[40] B. Norris and B. Demsky. CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *OOPSLA*, pages 131–150, 2013.

[41] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *PPoPP*, pages 167–178, 2003.

[42] J. Pichon-Pharabod and P. Sewell. A Concurrency Semantics for Relaxed Atomics that Permits Optimisation and Avoids Thin-Air Executions. In *POPL*, pages 622–633, 2016.

[43] E. Pozniansky and A. Schuster. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *CCPE*, 19(3):327–340, 2007.

[44] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, pages 27–37, 1997.

[45] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, pages 11–21, 2008.

[46] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[47] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.

[48] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, pages 15–26, 2011.

[49] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[50] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/FSE*, pages 205–214, 2007.

[51] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*, pages 221–234, 2005.