Instrumentation Bias for Dynamic Data Race Detection*

BENJAMIN P. WOOD, Wellesley College, USA
MAN CAO, Google Inc., USA
MICHAEL D. BOND, Ohio State University, USA
DAN GROSSMAN, University of Washington, USA

This paper presents Fast Instrumentation Bias (FIB), a sound and complete dynamic data race detection algorithm that improves performance by reducing or eliminating the costs of analysis atomicity. In addition to checking for errors in target programs, dynamic data race detectors must introduce synchronization to guard against *metadata races* that may corrupt analysis state and compromise soundness or completeness. Pessimistic analysis synchronization can account for nontrivial performance overhead in a data race detector.

The core contribution of FIB is a novel cooperative ownership-based synchronization protocol whose states and transitions are derived purely from preexisting analysis metadata and logic in a standard data race detection algorithm. By exploiting work already done by the analysis, FIB ensures atomicity of dynamic analysis actions with zero additional time or space cost in the common case. Analysis of temporally thread-local or read-shared accesses completes safely with no synchronization. Uncommon write-sharing transitions require synchronous cross-thread coordination to ensure common cases may proceed synchronization-free.

We implemented FIB in the Jikes RVM Java virtual machine. Experimental evaluation shows that FIB eliminates nearly all analysis atomicity costs on programs where data often experience windows of thread-local access. Adaptive extensions to the ownership policy effectively eliminate high coordination costs of the core ownership protocol on programs with high rates of serialized sharing. FIB outperforms a naïve pessimistic synchronization scheme by 50% on average. Compared to a tuned optimistic metadata synchronization scheme based on conventional fine-grained atomic compare-and-swap operations, FIB is competitive overall, and up to 17% faster on some programs. Overall, FIB effectively exploits latent analysis and program invariants to bring strong integrity guarantees to an otherwise unsynchronized data race detection algorithm at minimal cost.

69

1 INTRODUCTION

A *data race* in a shared-memory multithreaded program execution is a pair of memory accesses to the same shared-memory location, by different threads, unordered by synchronization, where at

---

Authors' addresses: Benjamin P. Wood, Wellesley College, USA, benjamin.wood@wellesley.edu; Man Cao, Google Inc. USA, manc@google.com; Michael D. Bond, Ohio State University, USA, mikebond@cse.ohio-state.edu; Dan Grossman, University of Washington, USA, djg@cs.washington.edu.

---

least one access is a write. Data races execute in any order, yielding unpredictable program state and confusing errors that are difficult to reproduce. Data races often factor into more complex shared-memory errors such as violations of atomicity or determinism. Memory consistency models of modern mainstream programming languages [Boehm and Adve 2008; Manson et al. 2005] guarantee that programs free of data races execute with the intuitive semantics of *sequential consistency* [Adve and Gharachorloo 1996; Adve and Hill 1990; Lamport 1979], but they make weak or no guarantees about programs with data races [Adve and Boehm 2010].

1.1   Data Race Detection Is Necessary but Slow.

Accurate data race detection is necessary. Programming on a weak memory model with no visibility into its behavior is a recipe for error-ridden code. Nonetheless, mainstream memory models fail to support programmers' reasoning efforts exactly when they need help [Adve and Boehm 2010; Boehm and Adve 2012]. Programmers need accurate tools to reason directly about the presence or absence of data races and other concurrency bugs in programs. Data race freedom or accurate data race detection is a prerequisite for many analyses of higher-level multithreading safety properties, such as atomicity [Flanagan et al. 2008; Shpeisman et al. 2007] and determinism [Bergan et al. 2010; Devietti et al. 2009; Olszewski et al. 2009].

Several researchers have further proposed *data race exceptions*, to enforce data race freedom at run time and simplify multithreaded memory semantics with strong guarantees in all cases [Adve 2010; Ceze et al. 2009; Elmas et al. 2007; Lucia et al. 2010; Marino et al. 2010; Wood et al. 2014]. On the second in a pair of accesses that race, such a system raises an exception instead of completing the access. Exceptions make data races obvious at run time like null pointer dereferences, but they are useful only if accurate and fast. *Accurate* means *sound* (no missed races) and *complete* (no false races) over an execution.[1] Sound static data race detectors conservatively report false races and are not suited for run-time data race exceptions [Flanagan and Freund 2000; Naik et al. 2006]. *Data race detection* means *dynamic data race detection* in this paper unless noted.

Accurate data race detection is slow. Several software [Choi et al. 2002; Christaens and Bosschere 2001; Effinger-Dean et al. 2012; Savage et al. 1997; Serebryany and Iskhodzhanov 2009; Yu et al. 2005] and hardware [Min and Choi 1991; Muzahid et al. 2009; Prvulovic 2006; Prvulovic and Torrellas 2003; Zhou et al. 2007] best-effort data race detectors attain reasonable performance by algorithms or optimizations that sacrifice accuracy. Other tools accurately detect data races that can violate sequential consistency or related consistency properties with relatively low overhead [Biswas et al. 2015; Lucia et al. 2010; Marino et al. 2010; Singh et al. 2011], but they do not detect all data races. Sound and complete data race detection systems require unimplemented hardware support [Devietti et al. 2012; Peng et al. 2017; Wood et al. 2014] or high run-time overheads [Elmas et al. 2007; Flanagan and Freund 2009, 2013]. For example, the state-of-the-art accurate software data race detector, FastTrack, slows program execution by several times [Flanagan and Freund 2009, 2017; Rhodes et al. 2017].

1.2   Software Data Race Detectors Require Defensive Synchronization.

In general, *analysis barriers* inserted immediately before each memory access in the target program must atomically check and update analysis metadata stored in shared memory. To preserve sound-ness and completeness, a dynamic data race detector must insert additional synchronization to

---

[1]This paper follows terminology conventions of data race detection research that can support data race exceptions [Devietti et al. 2012; Elmas et al. 2007; Flanagan and Freund 2009; Peng et al. 2017; Wood et al. 2014] with respect to soundness and completeness. Data race detectors are *accepters of data-race-free executions*. *Sound* means *accepts only* data-race-free executions; *complete* means *accepts all* data-race-free executions.

defend the analysis metadata used by these barriers against *metadata races*. The literature tends to treat the presence or absence of metadata synchronization as an artifact of implementation, despite the potential impacts on soundness, completeness, and performance.

Pessimistic synchronization of metadata can be expensive, accounting for 20-90% of the run-time overhead of a FastTrack [Flanagan and Freund 2009] implementation, according to experiments in prior work [Devietti et al. 2012]. Our own experiments show that an implementation of FastTrack in the Jikes RVM Java virtual machine [Alpern et al. 1999] using pessimistic spin locks for analysis atomicity can run over 16 times slower, and about 107% slower on average, than an unsynchronized implementation. An implementation of FastTrack using optimistic synchronization with atomic compare-and-swap (CAS) operations for shared metadata updates runs 8% slower than an unsynchronized version on average. Although the average synchronization overhead in our experiments is rather low: (1) the overhead is high for some programs; (2) prior work has reported higher synchronization overhead; and (3) prior synchronization schemes have generally been applied without fully integrating with the analysis to exploit its core invariants.

1.3 Cooperative Analysis Atomicity Mitigates Synchronization Costs.

The main contribution of this paper is *Fast Instrumentation Bias* (FIB, §5), a novel cooperative protocol for accurate data race detection that often reduces – and in many cases eliminates – the cost of analysis atomicity. FIB provides analysis atomicity for no added cost on common-case thread-local and read-shared accesses by introducing safe synchronous coordination between threads in uncommon cases. Shifting blanket pessimistic synchronization overhead to rare cooperative events can reduce overall cost, an insight whose foundations are exploited in diverse ways by cooperative systems such as cache coherence [Papamarcos and Patel 1984], biased locking [Bacon et al. 1998; Kawachiya et al. 2002; Nakaike and Michael 2010; Pizlo et al. 2011; Rajwar and Goodman 2001; Rogers and Iyengar 2011; Russell and Detlefs 2006; Vasudevan et al. 2010], the Octet dependence tracker [Bond et al. 2013] and other cooperative techniques for object-granularity race detection [von Praun and Gross 2001], and the RADISH hardware-supported data race detector [Devietti et al. 2012]. Unlike these systems, most status information and checks used by the FIB cooperative protocol incur no extra cost, since they are derived from existing data race detection metadata and logic. Further, FIB tracks data races and ownership status at a fine grain: per field rather than per object.

A secondary contribution of this paper is *FastTrack-Ownership (FTO)*, a modification of the FastTrack data race detection algorithm that is suitable for FIB to extend. FTO and FIB are natural extensions to the FastTrack's algorithm [Flanagan and Freund 2009] that derive *ownership* for each memory location from FastTrack's access history metadata. Ownership states grant threads permission for certain analysis operations. Non-owner threads initiate cooperative ownership state transitions to gain analysis permission. Owner threads perform synchronization-free analysis operations with the guarantee that non-owner threads may interfere only at well-defined points where the owner thread explicitly cooperates. Ownership checks on FIB fast paths for temporally thread-local or read-shared accesses are subsumed by FTO's preexisting analysis actions. Compared to an unsynchronized version of the analysis, *FIB enforces atomicity of analysis barriers with zero time overhead in common-case barriers and zero per-location storage overhead in all cases*.

The core FIB protocol is suited to programs with high rates of temporally thread-private and concurrent read-shared data accesses. Under high rates of serialized sharing, the protocol can trigger expensive coordination for ownership state transitions too frequently. We extend the core FIB protocol with two adaptive optimizations. *Predictive read sharing* predicts upcoming read sharing with a simple heuristic to avoid coordination costs for a reactive ownership state transition by making the transition preemptively at low cost. *Adaptive fail-over to local CAS-based synchronization* handles frequent serialized write sharing without the high cost of repeated coordination.

We implemented FIB and alternative schemes for analysis atomicity in the Jikes RVM [Alpern et al. 1999] Java virtual machine (§7). Experimental evaluation on a range of multithreaded Java programs shows that FIB's cooperative protocol can achieve analysis atomicity at low cost on many programs (§8). FIB introduces little or no cost relative to unsynchronized analysis on a majority of the 10 programs. Adaptive extensions reduce overheads of the core FIB protocol on serialized sharing by up to 26%. FIB outperforms a naïve pessimistic synchronization scheme by 50% on average. Compared to a tuned optimistic metadata synchronization scheme based on fine-grained atomic compare-and-swap operations, FIB is competitive on average, and up to 17% faster on some programs. Overall, FIB effectively exploits latent analysis and program invariants to bring strong integrity guarantees to an otherwise unsynchronized data race detection algorithm at minimal cost.

1.4 Outline

This paper is organized as follows:

§2 reviews foundations in data race detection.

§3 introduces *FastTrack-Ownership (FTO),* an accuracy-preserving revision to FastTrack that serves as a basis for FIB.

§4 surveys implementations of analysis atomicity and cooperative resource control.

§5 presents the key contribution, *Fast Instrumentation Bias (FIB),* a cooperative protocol for accurate data race detection that provides analysis atomicity at zero cost in the common case.

§6 describes two extensions to FIB for adaptive derivation of access history ownership to reduce high communication costs for programs with extensive serialized sharing.

§7 describes our implementation of FIB and several variations on FastTrack-Ownership in the Jikes RVM Java virtual machine.

§8 presents experimental performance and profiling evaluation of FIB against competing schemes for analysis atomicity.

§9 discusses related work.

§10 concludes.

2 BACKGROUND: THE FASTTRACK DATA RACE DETECTION ANALYSIS

This section reviews foundations of dynamic data race detection with FastTrack [Flanagan and Freund 2009].

2.1 Happens-Before and Data Races

The *happens-before* relation ($<_T$) is a reflexive partial order over operations in an execution trace T due to sequencing within threads and synchronization across threads [Lamport 1978]. A *data race* is a pair of *concurrent, conflicting* memory accesses [Netzer and Miller 1992]. Two operations a and b are *concurrent* in trace T if and only if they are not ordered by the happens-before relation (neither $a <_T b$ nor $b <_T a$). Concurrent accesses must be by separate threads. Two accesses *conflict* if they access the same location and at least one of the accesses is a write.

2.2 Tracking Logical Time in FastTrack

FastTrack [Flanagan and Freund 2009] tracks the happens-before relation with a system of logical time recorded by *epochs* and *vector clocks.* Figure 1 summarizes the syntax of FastTrack analysis metadata, explained in this section and §2.3. An *epoch,* $e = c@t$, is a logical time, c, local to a single thread, t. A *vector clock* [Fidge 1991; Mattern 1989], v, is a mapping from each thread, t, to an integer logical clock, c, together representing a frontier in happens-before order. We use vector clocks interchangeably with sets of epochs of unique threads; a vector clock lookup returns an

Location	x	
Thread ID	t, u	
Operation	a, b	$::= \mathsf{w}_t x \mid \mathsf{r}_t x \mid \dots$
Trace	T	$::= \text{Operation}^*$

Clock	c	$: \mathbb{Z}^+$
Epoch	$e, c@t$	$: \text{Clock} \times \text{Thread ID}$
VC	v	$: \text{Thread ID} \rightarrow \text{Epoch}$
Thread VCs	C	$: \text{Thread ID} \rightarrow \text{VC}$
Lock VCs	L	$: \text{Lock} \rightarrow \text{VC}$
Last Reads	R	$: \text{Location} \rightarrow (\text{Epoch} \cup \text{VC})$
Last Writes	W	$: \text{Location} \rightarrow \text{Epoch}$

Fig. 1. Syntax and structure of analysis traces and metadata.

epoch: $v(t) = c@t$. The happens-before relation is captured by ordering over epochs and vector clocks. Epoch $c@t$ happens before epoch $d@u$, written $c@t \preceq d@u$, if and only if $t = u$ and $c \leq d$. Epoch $c@t$ happens before vector clock v, written $c@t \preceq v$, if and only if $c@t \preceq v(t)$. Vector clock v_1 happens before vector clock v_2, written $v_1 \sqsubseteq v_2$, if and only if $\forall t, v_1(t) \leq v_2(t)$. The vector-clock merge operation $v_1 \sqcup v_2$ computes the element-wise maximum of two vector clocks:

$$v_1 \sqcup v_2 \equiv \{d@t \mid c_1@t = v_1(t) \wedge c_2@t = v_2(t) \wedge d = max(c_1, c_2)\}$$

To track the happens-before ordering effects of synchronization, FastTrack maintains a vector clock, C_t, for each thread t, representing the last logical time (epoch) in each thread that happened before the current operation of thread t. Thread t's entry in its own vector clock, $C_t(t)$, is its current epoch. Each lock, l, is tracked by a vector clock, L_l, representing the last logical time (epoch) in each thread that happened before the last release of lock l.

Initially, each thread starts in its own first epoch, with no knowledge of other threads' initial epochs: $C_t = \{0@u \mid u \neq t\} \cup \{1@t\}$. Each lock starts with an empty vector clock: $C_t = \{0@u\}$. When thread t acquires lock l, all events that happened before the last release of lock l are also guaranteed to happen before all future events in thread t. Accordingly, FastTrack updates thread t's vector clock to show this happened-before relationship: $C_t := C_t \sqcup L_l$. When thread t releases lock l, all events that happened before this release are also guaranteed to happen before all future acquires of this lock by any thread. Accordingly, FastTrack updates lock l's vector clock to show this happened-before relationship: $L_l := C_t$. Since this records thread t's current logical time, FastTrack subsequently increments thread t's current epoch, $C_t(t)$, to show that any future events in thread t happen at a new logical time that did not happen before the release of lock l. Other types of synchronization are tracked similarly.

This work focuses solely on access tracking, using standard techniques for vector-clock synchronization tracking as is. For additional detail and discussion of synchronization tracking, refer to [Flanagan and Freund 2009].

2.3 Tracking Accesses in FastTrack

In data-race-free traces, T, all writes to a location, x, must be ordered by happens-before with respect to all other accesses to x. Reads of x in separate threads may be mutually concurrent in T, so long as they are ordered with all writes to x. FastTrack checks data race freedom of a program execution incrementally, testing each operation in turn to determine if it races with any earlier operation. Due to the transitivity of happens-before and the goal of detecting the first data race [Adve et al. 1991; Banerjee et al. 2006; Choi and Min 1991; Flanagan and Freund 2009; Min and Choi 1991; Netzer and Miller 1991, 1992] for data race exceptions, it suffices to check each operation for races against only the last physically earlier write by any thread and the last physically earlier read in each thread that follows the last write by any thread. When all such reads are ordered, the check is even simpler [Flanagan and Freund 2009].

Table 1. FastTrack analysis barrier logic based on the access history (R_x, W_x) for data location x, as in [Flanagan and Freund 2009]. The notation $0@t_0$ denotes the lack of any last read, represented by the origin epoch in our formulation.

Event	Case Check	Order Check	Update	FastTrack Rule
$r_t x$	$R_x = C_t(t)$			FT READ SAME EPOCH
	$R_x = c@u$	$R_x \preceq C_t \wedge W_x \preceq C_t$	$R_x := C_t(t)$	FT READ EXCLUSIVE
		$W_x \preceq C_t$	$R_x := \{c@u, C_t(t)\}$	FT READ SHARE
	$R_x = v$	$W_x \preceq C_t$	$R_x(t) := C_t(t)$	FT READ SHARED
$w_t x$	$W_x = C_t(t)$			FT WRITE SAME EPOCH
	$R_x = c@u$	$R_x \preceq C_t \wedge W_x \preceq C_t$	$W_x := C_t(t)$	FT WRITE EXCLUSIVE
	$R_x = v$	$R_x \sqsubseteq C_t \wedge W_x \preceq C_t$	$R_x := 0@t_0, W_x := C_t(t)$	FT WRITE SHARED

2.3.1 FastTrack Access History. FastTrack maintains an *access history*, H, for each location. x. The access history summarizes accesses to x in a trace, T, by the epochs of the last accesses to x in T. The access history for x in T is a collection, H, of epochs. Each epoch in the history satisfies the property that, for all accesses, a, to location x in trace T, access a occurred in an epoch, e, that happened before or in some epoch in the access history: $\forall$ such access epochs $e, e \in H \vee \exists e', e' \in H \wedge e \preceq e'$

Each access history representation has two parts:

- The **Last Write,** W_x, tracks the epoch of the last write to location x by any thread in trace T. In the absence of writes to x, the origin epoch, $0@t_0$, is used.
- The **Last Read(s),** R_x, tracks, for each thread, t, the epoch of the last read to x by t. When thread t's last read of x happens before some other other access to x, its epoch may be omitted. The last reads value R_x thus takes one of two forms:
  - A read epoch, e, when all past reads happen in or before e and a read of x occurred in e. The origin epoch, $0@t_0$, notated $\perp_e$ in [Flanagan and Freund 2009], is used when there is no last read.
  - A *read map* (or vector clock), v, where $v(t)$ is the epoch of t's last read of x, when concurrent reads of x have occurred since the last write of x.

2.3.2 FastTrack Access Analysis Barriers. Table 1 shows the FastTrack analysis barrier logic for read and write events from [Flanagan and Freund 2009]. Analysis cases are organized by read ($r_t x$) and write ($w_t x$) operations and labeled by name on the right. Each analysis case is broken into three components that check or update the access history (R_x, W_x) for data location x. The **Case Check** uses the access history to identify which case of the analysis to deploy. The **Order Check** gives a predicate that must be true on the access history to establish sufficient happens-before ordering with accesses of other threads to make the current access data-race-free under the given case. If this predicate is false, FastTrack reports a data race. The **Update** column shows how FastTrack updates the access history to record a data-race-free access.

2.3.3 FastTrack Read Barrier. A read to location x by thread t is safe if all writes to x so far happen before thread t's current epoch. FastTrack has the following read cases, shown in Table 1:

- FT READ SAME EPOCH: If the last read is the current epoch, then this read is DRF, since a read has already been checked and recorded in this epoch. Any further check or update would be redundant.
- If the last read happened in a different epoch, then there are two conditions under which the current read may be DRF:

- FT READ EXCLUSIVE: If both the last read and write happened before the current logical time, then the current read is DRF and supersedes the last read in the access history.
  - FT READ SHARE: If the last write happened before the current logical time, but the last read is concurrent, then the current read is DRF. The access history must retain the epochs of both last and current reads as a read map (vector clock) to check future writes.
- FT READ SHARED: If there are multiple concurrent last reads since the last write, then the current read is DRF if the last write happens before it. The current epoch is recorded among the set of concurrent last reads to check future writes.

In all other cases, the read is part of a data race.

2.3.4 Write Barrier. A write to location x by thread t is safe if all accesses to x so far happen before thread t's current epoch. FastTrack checks three cases:

- FT WRITE SAME EPOCH: As with reads, if the last write was in the current epoch, any further analysis is redundant.
- FT WRITE EXCLUSIVE: If the last read is an epoch, then the current write is DRF if both the last read and the last write happen before the current logical time. The current epoch is recorded as the last write for checking future accesses.
- FT WRITE SHARED: If the access history encodes multiple concurrent last reads, then the current write is DRF if all accesses in the history (write and reads) happened before the current logical time. The current epoch is recorded as the last write for checking against future accesses. The last reads are erased, since they all happened before the new last write. Future accesses that race with one of the last reads also race with the new last write and will be detected.

In all other cases, the write is part of a data race.

3 REVISING FASTTRACK FOR OWNERSHIP

To support FIB's ownership protocol and optimize for minimal analysis fast paths, we make a novel set of small accuracy-preserving specializations to the basic FastTrack (FT) access analysis barriers. The resulting new analysis, *FastTrack-Ownership (FTO)*, maintains stronger invariants on access history to make a notion of *thread ownership* manifest in access history metadata. This secondary contribution simplifies analysis common cases and forms a foundation for FIB (§5).

FastTrack-Ownership access history invariants. FTO maintains and exploits the following invariants on access histories relative to a program trace, T, representing the execution so far:

INVARIANT 1. *As in FastTrack, all writes of x in T and all reads of x occurring at an earlier physical time than the last write to x in T logically happen before the last write epoch W_x.*

The original FastTrack algorithm enforces and exploits Invariant 1 to erase epochs of earlier writes and some earlier reads in its access history updates for write operations.

INVARIANT 2. *Each read and write of x in T happens before at least one last read epoch in R_x.*

FTO introduces this invariant, which strengthens a weaker invariant in the original FastTrack algorithm and exploits FastTrack's reliance on the last read(s) information for all case checks. FastTrack maintains a similar invariant when R_x is a read map, but not when it is a single epoch [Flanagan and Freund 2009]. Specifically, FastTrack's write barriers may leave last reads that are empty or older than the recorded last write epoch.

Table 2. FastTrack-Ownership (FTO) analysis barrier logic. Case rows are grouped by the nature of order checks and updates to be performed. Rules marked "FT" are unchanged from Table 1. Rules marked "∗" are changed. Rules marked "+" are added.

Event	Owner Check	Order Check	Update	FTO Rule
$r_t x$	$R_x = C_t(t)$			FT READ SAME EPOCH
	$R_x = v \wedge C_t(t) \in v$			+ READ SHARED SAME EPOCH
$w_t x$	$W_x = C_t(t)$			FT WRITE SAME EPOCH
$r_t x$	$R_x = c@t$		$R_x := C_t(t)$	+ READ OWNED
	$R_x = v \wedge c@t \in v$		$R_x(t) := C_t(t)$	+ READ SHARED OWNED
$w_t x$	$R_x = c@t$		$R_x := C_t(t), W_x := C_t(t)$	+ WRITE OWNED
$r_t x$	$R_x = v$	$W_x \preceq C_t$	$R_x(t) := C_t(t)$	FT READ SHARED
$r_t x$	$R_x = c@u \wedge u \neq t$	$R_x \preceq C_t$	$R_x := C_t(t)$	∗ READ EXCLUSIVE
		$W_x \preceq C_t$	$R_x := \{c@u, C_t(t)\}$	FT READ SHARE
$w_t x$	$R_x = c@u \wedge u \neq t$	$R_x \preceq C_t$	$R_x := C_t(t), W_x := C_t(t)$	∗ WRITE EXCLUSIVE
$w_t x$	$R_x = v$	$R_x \sqsubseteq C_t$	$R_x := C_t(t), W_x := C_t(t)$	∗ WRITE SHARED

FastTrack-Ownership access analysis barriers. Table 2 shows the analysis barrier logic for FTO in the same style as Table 1 does for FastTrack. Rules are marked "FT", "∗", or "+" to distinguish unchanged FastTrack rules, modified rules, and new rules, respectively.

To maintain Invariant 2, all FTO write barriers that update the last write must also update the last read to be identical. While the "last read" would thus be better named the "last read or write," we retain the "last read" term for consistency with FastTrack. Invariant 2 ensures that the last read(s) component of an access history alone suffices both to determine the overall last access(es) to a location and to check DRF for a new access. This invariant simplifies *order checks* in many cases, at the cost of extra update work on write cases.

FTO's modified WRITE EXCLUSIVE and READ EXCLUSIVE cases now require only one order check: $R_x \preceq C_t$. Invariant 2 and the transitivity of happens-before establish that if the predicate $R_x \preceq C_t$ holds, then the original FastTrack's second order check, $W_x \preceq C_t$, also holds.

Additionally, Invariant 2 enables the new special cases WRITE OWNED and READ OWNED. If there is a single last read epoch and it is an epoch of the current thread – even if it is not the current epoch – it must happen before the current epoch. Transitively, by Invariant 2, the last write and all other accesses also happened before the current epoch. The new READ SHARED OWNED applies similar logic to the READ SHARED case: if there is a last reads map containing an entry for the current thread, an earlier access under the READ SHARED case established ordering with the last write, so there is no need to check it here, again by Invariant 2. Thus these special cases of the EXCLUSIVE and READ SHARED rules require no order checks.

Finally, FTO introduces two specializations that are possible even in FastTrack. The new READ SHARED SAME EPOCH case applies the same-epoch optimization for read map entries to avoid redundant updates. This case is also used by [Flanagan and Freund 2017] and the implementation of FastTrack in [Biswas et al. 2015]. The WRITE SHARED case elides an order check against the last write, since it is established transitively through order checks with the last reads, even without Invariant 2.

4 BACKGROUND & MOTIVATION: IMPLEMENTING ATOMIC ANALYSIS BARRIERS

This section surveys implementations of atomic analysis barriers for dynamic data race detection.

Data race detection analysis barriers must execute atomically before the accesses they analyze to ensure soundness. It is easy to see that interleaved checks and updates from the read or write barriers in Tables 1 and 2 could lead to lost updates and missed data races. An ideal implementation of atomic barriers satisfies two requirements. Ensuring atomicity should ideally impose little or no time overhead beyond analysis logic and little or no space overhead beyond the access history.

Pairing each access history with a lock to protect its analysis barriers enforces atomicity, but introduces pessimistic locking costs on every access and requires access history storage for a lock. An implementation in Jikes RVM runs on average 107% slower than unsynchronized FastTrack-Ownership (FTO) on DaCapo benchmarks [Blackburn et al. 2006] and up to over 16 times slower in some cases.

4.1 Optimistic Synchronization

FastTrack and FTO can also support an optimistic approach to analysis atomicity using compare-and-swap (CAS). Read and write same-epoch fast paths may execute using a single unsynchronized memory load and check, assuming proper ordering restrictions in the compiler and hardware. In practice, achieving this ordering is relatively inexpensive on comparatively strong mainstream hardware memory consistency models, such as TSO [Adve and Gharachorloo 1996; Sewell et al. 2010], but may carry greater expense on weaker memory models. Other common analysis paths with access history updates require only a CAS and a few unsynchronized memory operations in the common, uncontended case. The remaining rare, complicated, or contended cases can fall back on lightweight spin locks implemented by reserving a special locked value for the last read or last write field of the access history.

Figure 2 shows simplified pseudocode for a fine-grained optimistic scheme for atomicity. Standard non-updating fast paths, not shown here, are executed without synchronization. This pair of barriers represents a middle ground. They employ fine-grained CAS operations on individual read map entries to avoid serializing access to an existing read map, but they use simpler limited mutual exclusion in cases that are not expected to appear concurrently (*e.g.*, writes) or where allowing concurrency is not productive (*e.g.*, writes or the FT READ SHARE transition, §2.3, §3).

In this pseudocode, CAS success provides full memory ordering. Release fences precede stores to x.R that act as lock releases and interact with CASes to order operations on x.W as well. The casHB function shown supports the FT WRITE SHARED case (§2.3, §3), when x.R refers to a map of concurrent last reads. FTO checks ordering against map entries, discards the map, and stores an epoch for x.R. casHB locks map entries permanently, since the entries will not (and should not) be used again for future analysis: well-ordered future barriers cannot observe this map; racing write barriers cannot observe this map, due to mutual exclusion on x.R; racing read barriers that observe this map detect the LOCKED entry (and the race) when CASing to update.

An implementation in Jikes RVM runs on average 6% slower than unsynchronized FTO on DaCapo benchmarks [Blackburn et al. 2006] and up to 32% slower in some cases. These tuned but conventional optimistic approaches achieve zero space overhead and lower time overhead than naïve pessimistic synchronization, but still require preemptive synchronization on many operations where it is wasted, even if it is only the cost of a single CAS operation.

4.2 Cooperative Synchronization

Systems for cooperative concurrency can help avoid wasted synchronization cost on cases where it is rarely needed in practice, but existing systems fall short of our criteria or achieve them only with unavailable hardware support. Guarding access history with biased locks (§9.1) could improve performance for uncontended thread-local sections, but adds space overhead and at least some time overhead for locking even in uncontended cases. Wrapping analysis barriers in

Metadata record types for analysis barriers

```
Thread t {                    AccessHistory x {
  VC  C   ≡ C_t                 Epoch or Map  R  ≡ R_x
}                               Epoch  W          ≡ W_x
                              }
```

Read analysis barrier for $r_t x$

```
read(Thread t, AccessHistory x) {
  // READ SAME EPOCH
  if (x.R == t.C[t]) return SAFE

  // CAS one fine-grained map entry
  // if possible to reduce contention.
  Epoch or Map  r := x.R
  if (isMap(r)) {        // READ SHARED
    Epoch e := r[t]
    if (e != LOCKED) {
      Epoch w := x.W
      if (w <= t.C[thread(w)]
          && CAS(&r[t], e, t.C[t])) {
        return SAFE
      }
    }
  }

  // Slow Path: lock last reads
  r := lock(x)
  w := x.W
  if (w <= t.C[thread(w)]) {
    if (isEpoch(r)) {
      if (r <= t.C[thread(r)]) {
        r := t.C[t]   // READ EXCLUSIVE
      } else {
        r := {r, t.C[t]}  // READ SHARE
      }
    } else {          // Map created
      r[t] := t.C[t]  // concurrently
    }                 // READ SHARED
    release fence
    x.R := r          // unlock
    return SAFE
  }
  return RACE
}
```

Write analysis barrier for $w_t x$

```
write(Thread t, AccessHistory x) {
  // WRITE SAME EPOCH
  if (x.W == t.C[t]) return SAFE

  // Lock the last reads word, then
  // check its parts.
  Epoch or Map  r := lock(x)
  // WRITE EXCLUSIVE
  if (isEpoch(r) && r <= t.C[thread(r)]
          // WRITE SHARED
          || isMap(r) && casHB(x, t, r)) {
    // Update and release if SAFE
    x.W := t.C[t]
    release fence
    x.R := t.C[t]
    return SAFE
  }
  return RACE
}

casHB(AccessHistory x, Thread t, Map map) {
  // Check & lock each entry.
  // If CAS fails, there is a race.
  for (u in map) {
    Epoch e := map[u]
    if (e > t.C[u]
        || !CAS(&map[u], e, LOCKED)) {
      return false
    }
  }
  return true
}

lock(AccessHistory x) {
  do {
    Epoch or Map  old := x.R
  } while (!CAS(&x.R, old, LOCKED))
  return old
}
```

Fig. 2. CAS-based FastTrack-Ownership analysis barriers for $r_t x$ and $w_t x$ based on high-level definitions shown in Table 2. The pseudocode here does not optimize OWNED cases separately from EXCLUSIVE cases.

transactions with commodity hardware transactional memory has shown promising reduction in overhead for C/C++ programs, but absolute overheads remained high [Matar et al. 2014]. Object race detection [von Praun and Gross 2001] used a cooperative system to filter out redundant data race checks. Octet [Bond et al. 2013; Cao et al. 2016] tracks cross-thread dependences efficiently using an optimistic cooperative protocol. However, object granularity does not suffice for accurate fine-grained data race detection. While none of these alternatives is a clear win over the more pessimistic software solutions described above, cooperative synchronization is a promising model for implementing atomic barriers.

5 THE FIB PROTOCOL

This section describes *Fast Instrumentation Bias (FIB)*, an accurate dynamic data race detection algorithm that implements a cooperative protocol for analysis atomicity based on FastTrack-Ownership (§3). In practice, most accesses involve data that are temporally thread-local or read concurrently by multiple threads, with no writes. Threads access data last accessed by a separate thread relatively rarely. This pattern motivates FIB's cooperative approach to atomicity, in which analysis of temporally thread-local and read-shared accesses proceeds synchronization-free, while analysis of accesses involving rare cross-thread interaction bears the cost of atomicity through synchronous coordination.

FIB derives an *ownership state* directly from each access history without additional storage. This ownership state dictates which threads have permission to use and change the access history without coordination. Each analysis barrier checks ownership state to determine what (if any) coordination is required to ensure the barrier is atomic. The first step of a data race check subsumes the ownership check. In the common case, this joint check shows both that the current thread has exclusive or shared ownership of the location – and can thus proceed without coordination – and that there is no data race. In the uncommon case, the ownership state grants insufficient permission, so the thread must request that the current owner thread(s) perform a state transition and data race check on its behalf.

Threads respond to incoming requests at well-defined yield points. Since ownership state transition requests are cooperative, not preemptive, owner threads are guaranteed that no other threads interfere in the access history until the owner thread responds explicitly to a transition request. This guarantee supports safe, synchronization-free analysis barriers for owner threads.

FIB adds zero time overhead to common-case analysis barriers, unifying synchronization-free ownership checks and data race checks under compatible ownership states. FIB adds zero space overhead in access histories, deriving ownership state purely from access history. The remainder of this section summarizes the FIB ownership protocol (§5.1) and follows with in-depth discussion of local (§5.2) and conflicting (§5.3, §5.4) transitions.

5.1 Ownership States and Transitions

Each per-location access history inhabits one of two ownership states that grant certain permissions to check or update the access history:

- Excl(t) grants thread t exclusive permission to check and update all parts of the access history without synchronization.
- Shared(S) grants every thread $t \in S$ permission to check all parts of the access history and update thread t's entry in the access history read map without synchronization.

Ownership of the access history (§2.3.1, §3) for a data location, x, is derived purely from by the value of its last-reads component, R_x:

Table 3. Summary of FIB ownership state transitions and corresponding FastTrack-Ownership analysis cases. The set of threads with non-empty entries in a vector clock or read map, v, is denoted $v_{tids} \equiv \{t \mid c@t \in v\}$. Inter-thread communication by thread t with thread u is denoted $t \leftrightarrow u$.

Type	Op	FTO Analysis			FIB Transition		
		Owner Check	Order Check	Update	Start State	Coordinate	End State
pure local	$r_t x$	$R_x = C_t(t)$			$Excl(t)$		$Excl(t)$
		$R_x = v$ $C_t(t) \in v$			$Shared(v_{tids})$		$Shared(v_{tids})$
	$w_t x$	$R_x = C_t(t)$			$Excl(t)$		$Excl(t)$
impure local	$r_t x$	$R_x = c@t$		$R_x := C_t(t)$	$Excl(t)$		$Excl(t)$
		$R_x = v$ $c@t \in v$		$R_x(t) := C_t(t)$	$Shared(v_{tids})$		$Shared(v_{tids})$
	$w_t x$	$R_x = c@t$		$R_x := C_t(t)$ $W_x := C_t(t)$	$Excl(t)$		$Excl(t)$
fence	$r_t x$	$R_x = v$ $c@t \notin v$	$W_x \le C_t$	$R_x(t) := C_t(t)$	$Shared(v_{tids})$	fence	$Shared(v_{tids} \cup \{t\})$
single conflict	$r_t x$	$R_x = c@u$ $u \neq t$	$R_x \le C_t$ $W_x \le C_t$	$R_x := C_t(t)$ $R_x := \{c@u, C_t(t)\}$	$Excl(u)$	$t \leftrightarrow u$	$Excl(t)$ $Shared(\{u, t\})$
	$w_t x$	$R_x = c@u$ $u \neq t$	$R_x \le C_t$	$R_x := C_t(t)$ $W_x := C_t(t)$	$Excl(u)$	$t \leftrightarrow u$	$Excl(t)$
multi conflict	$w_t x$	$R_x = v$	$v \sqsubseteq C_t$	$R_x := C_t(t)$ $W_x := C_t(t)$	$Shared(v_{tids})$	CAS; fence; $\forall u \in v_{tids}$, $t \leftrightarrow u$	$Excl(t)$

- $Excl(t)$ if $R_x = c@t \wedge c > 0$: If the last-reads field holds a non-zero epoch belonging to thread t, then thread t has *exclusive ownership* of the access history.
- $Shared(S)$ if $R_x = v \wedge S = \{t \mid \exists c, c@t \in v\}$: If the last-reads field holds a read map v, then all threads with an entry in v hold *shared ownership* of the access history.

Table 3 and Figure 3 summarize the FIB state transitions that occur in step with each FastTrack-Ownership (FTO) analysis. Each row in Table 3 shows a case in FTO (left columns *owner check, order check, update*, matching Table 2), paired with the FIB transition (right columns *start state, coordination, end state*) that takes effect in cooperation with this analysis case. The *owner check* of each analysis case also serves to derive the FIB *start state* currently in effect for the access history. The *order check* is undertaken in step with the FIB *coordination* required to effect the analysis case and FIB transition safely. The analysis *update* also completes the FIB transition, yielding an access history from which the *end state* is derived.

Rows in Table 3 are grouped by the type of FIB transition. *Local* self-transitions correspond to data race checks against last accesses by the current thread, where data races are never possible. Analysis barriers by non-owner threads initiate *conflicting* state transitions to dispatch data race checks against last accesses by other threads, where data races may occur.[2] Transitions require one of four levels of coordination:

---

[2] *Conflicting* refers to *metadata* accesses. A single-conflict transition may step from a state $Excl(u)$ to another state, both states potentially due to read barriers. In this case, the two corresponding data read accesses may not conflict, but updates in the access history by the read analysis barriers do.
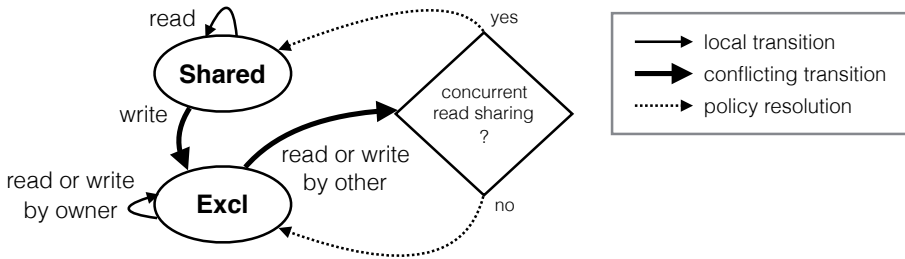
Fig. 3. FIB state transition overview.

- **Local (§5.2):** In state Excl(t) or Shared(S) s.t. $t \in S$, an analysis barrier by thread t executes *locally*, with a self-transition that requires no coordination and never detects data races.
- **Fence (§5.2, §5.4):** In state Shared(S), a read barrier by thread $t \notin S$ requires a store–load memory fence to coordinate a safe transition to state Shared($S \cup \{t\}$). The end state of this transition, unlike conflicting transitions, grants a monotonic increase over permissions granted by the start state, so coordination with current owners is unnecessary. The fence coordinates with potential concurrent write barriers, discussed in §5.4.
- **Single-conflict (§5.3):** In state Excl(u), a write or read analysis barrier by thread $t \neq u$ requires synchronous coordination with the single conflicting thread u to effect a safe transition to state Excl(t) or Shared($\{t, u\}$).
- **Multiple-conflict (§5.4):** In state Shared(S), a write analysis barrier by any thread t requires several synchronization steps to effect a safe transition to state Excl(t). This transition composes atomic compare-and-swap (CAS), store–load memory fence, and synchronous coordination with all threads $u \in S$.

Conflicting state transitions demand coordination with current owner threads, to ensure that these threads acknowledge invalidation of the current ownership state, and to ensure that the transition's data race check resolves against up-to-date access history potentially in concurrent use by current owner threads. Conflicting transitions require expensive synchronous cross-thread communication, but are rare in practice. In exchange, FIB common-case fast paths run free of overhead.

Figure 4 sketches a more concrete implementation for the logic of the read and write analysis barriers employed by FIB and described in the following sections. FIB derives its ownership state purely from the same access history metadata as FastTrack-Ownership (§3). FIB also adds some per-thread metadata to support cross-thread communication.

5.2 Local Transitions

Local self-transitions match the most common patterns of access in multithreaded programs: repeated accesses to temporally thread-local data and concurrent read-only access to shared data by multiple threads.

The read and write analysis barriers both start by loading the last read information, R_x, and checking if it is an epoch or a read map. This sequence serves identically as a lookup and check of ownership and a lookup and check for the first potential proof that the access is data-race-free.

5.2.1 Exclusive Reads and Writes. If the last read, R_x, is an epoch of this thread, then the access history is exclusive to this thread and the current access is data-race-free, by Invariant 2. Any updates in these barriers execute atomically even without synchronization, since other threads must coordinate with this thread before using the access history.

5.2.2　Shared Reads. Read barriers may also execute locally when the access history is Shared, *i.e.*, when there are multiple last reads represented by a read map. If there is no existing entry for a reading thread t in the read map, thread t has neither read since nor performed data race checks against the last write and must do so now (readShJoin in Figure 4). A store–load memory fence is necessary in this case to ensure potentially racing write barriers do not miss this read. Details of this coordination are discussed in §5.4. After the first read by thread t in Shared state, the presence of an entry for t in the read map indicates that the last write happens before an earlier read in this thread, so a new read in thread t is likewise ordered. No further check or coordination is needed.

5.3　Single-Conflict Transitions

A single-conflict transition is required when a thread t accesses a location whose last read, R_x, is an epoch of a different thread. When a barrier in thread t encounters an access history in state $\mathsf{Excl}(u)$, where $u \neq t$, thread t requests that thread u perform a data race check, update, and state transition on its behalf. We refer to this requested work as the *check-and-transfer* sequence. To support sending requests and responses between threads, each thread maintains a queue of incoming requests from other threads and a field to store an incoming response from another thread to a request of its own. If thread u is running, the request is enqueued with thread u and thread t waits for thread u to respond. If thread u is blocked, it is not actively using its ownership of x, so thread t can complete the check and transfer directly rather than sending a request and awaiting a reply.

When a thread reaches a *yield point* in execution, it processes all the requests in its queue and responds to each with the data race check result (not shown in Figure 4). Yield points must occur within a bounded interval to ensure forward progress. They are typically inserted in application code at calls, returns, loop back-edges, and blocking operations (including cross-thread requests described in this section). Managed language implementations already provide such yield points for run-time services such as garbage collection and on-stack replacement [Fink and Qian 2003].

Check and Transfer. Regardless of whether a request is served at a yield point in the current owner thread in response to a remote request or self-served by the requesting thread while the owner thread is blocked, the same check-and-transfer logic is applied. The standard data race check for this type of access suffices to determine if the transition is safe. The standard check updates the last read, R_x, as needed, thereby updating the ownership state and completing a transition.

5.4　Multiple-Conflict Transitions

Multiple-conflict transitions occur in write analysis barriers following concurrent reads in Shared state. Most read barriers by a thread t in Shared state never check for races with the last write, assuming the presence of an earlier entry for t in the Shared read map demonstrates that the last write, W_x, happens before an earlier access by thread t and therefore also happens before a new read by t, by Invariant 2 (§3). To support these common-case fast paths safely, Shared $\to$ Excl transitions, which update W_x and R_x, and $\mathsf{Shared}(S) \to \mathsf{Shared}(S \cup \{t\})$ transitions, which add new entries to read maps, must either preserve Invariant 2 (§3) or detect the first data race on x (after which the invariant is moot).

Writes in $\mathsf{Shared}(S)$ state are exceedingly rare (<0.00002% of barriers in most of the evaluated programs) so even a write barrier that communicates with all live threads would be feasible. The Fıв protocol requires coordination with only those threads in S (derived from the read map), scaling costs roughly with the breadth of read sharing preceding the write. The remainder of this section explains these transitions, as shown in readShJoin and writeSh (Figure 4).

Read and write barriers may race on the read sharing set S. Since a data write is involved, the existence of such a metadata race implies a data race in the target program. To avoid missing a

Metadata record types for analysis barriers

```
Thread t {
  VC  C   ≡ Cₜ
  ... // communication state
      // not shown
}
```

```
AccessHistory x {
  Epoch or Map  R   ≡ Rₓ
  Epoch  W          ≡ Wₓ
}
```

Read analysis barrier for $r_t x$

```
read(Thread t, AccessHistory x) {
  Epoch or Map  r := x.R
  if (isEpoch(r)
      && thread(r) == t)) {      // Excl(t) &&
    x.R := t.C[t]        // DRF by Invariant 2
    return SAFE
  } else if (isMap(r)) {
    if (r[t] != NONE) { // Shared({t,...}) &&
      r[t] := t.C[t]    // DRF by Invariant 2
      return SAFE
    } else {            // Shared(S), t not in S
      return readShJoin(r, t, x)
    }
  } else {              // Excl(u != t)
    return request(r, READ, t, x)
  }
}
```

```
readShJoin(Map r, Thread t, AccessHistory x) {
  r[t] := t.C[t]      // Shared(S)->Shared(S+t)
  store-load fence    // Ensure visible
  Epoch w := x.W
  if (w > t.C[thread(w)]) { // Check write race
    return RACE
  } else {
    return SAFE
  }
}
```

Write analysis barrier for $w_t x$

```
write(Thread t, AccessHistory x) {
  Epoch or Map  r := x.R
  if (isEpoch(r)) {
    if (thread(r) == t) {   // Excl(t) && DRF
      x.W := t.C[t]
      x.R := t.C[t]
      return SAFE
    } else {               // Excl(u != t)
      return request(r, WRITE, t, x)
    }
  } else {                 // Shared
    return writeSh(t, x, r)
  }
}
```

```
writeSh(Thread t, AccessHistory x, Map r) {
  if (!CAS(&x.R, r, t.C[t])) { // Shared->Excl(t)
    return RACE                // CAS fail = race
  } else {
    Epoch w := x.W
    x.W := t.C[t]              // Record write
    store-load fence           // Ensure visible
    for (_@u in t.C) {         // Coord. readers
      if (r[u] != NONE) {
        ack(u, t)              // Request/await ack
        if (r[u] > t.C[u]) { // Check race with u
          return RACE
        }
      }
    }
    return SAFE
  }
}
```

Fig. 4. Fɪʙ analysis barriers for $r_t x$ and $w_t x$ implementing the high-level definitions shown in Table 3, using the same metadata as defined in Figure 2. Pure same-epoch fast paths are omitted.

data race, the relevant cases in both barriers follow a pattern in reverse of the usual check–update order: (1) record the new access in the history, assuming no data race will occur; (2) ensure this update is visible to other threads with a store–load fence; then (3) perform data race checks and complete supporting coordination. If read and write analysis barriers race on Shared state, at least one observes the other's access history updates and reports the race.

5.4.1 Shared Write Barrier (`writeSh`*).* A write barrier in thread t finding a Shared(S) access history first uses a CAS operation to update R_x to its current epoch, making a preemptive transition to Excl(t) state, and then updates W_x to match. This is the only barrier case to break the ownership policy on R_x, so CAS failure implies a concurrent, racing write barrier. However, the preemptive ownership transition may temporarily break Invariant 2. Concurrent read barriers on x will eventually observe the new state, but some may miss it, assuming Invariant 2 and updating read map entries without synchronization. Shared reads are relatively common, so this rare case of the write barrier should carry the cost of coordination.

Next, a store–load fence ensures access history updates are visible, and the write barrier requests an *ack* from every thread in the read map, S. The ack communication guarantees that: (1) the responding thread will see the new Excl(t) state in future barriers on x; and (2) the requesting write barrier will check against the most recent read map entry from the responding thread. Logic for ack responses is not shown.

Finally, after receiving a thread's ack, the write barrier does the conventional happens-before check for that thread, comparing its entry in the read map to the corresponding entry in the writing thread's vector clock. The racing read barrier may detect the same data race.

5.4.2 Read Barrier Fence Transitions (`readShJoin`*).* Read barriers may race with the write barrier to add new entries in the read map in `readShJoin` and `writeSh`, respectively. The write barrier may miss these entries and the racing accesses they represent. To compensate, when a read barrier adds an entry for a previously absent thread to a read map, it follows this update with a fence. After ensuring its read map entry is visible, it performs a data race check against the last write, W_x. By performing update and then check, these cases of the read and write barriers work together to ensure at least one of the two detects the race if it occurs. The added cost of a fence in `readShJoin` case is inconsequential, considering it accounts for less than 0.03% of barriers.

6 ADAPTIVE POLICIES

This section introduces *predictive read sharing* (§6.2) and adaptive support for *serialized write sharing* (§6.3), two novel adaptive extensions to FIB's ownership derivation policy that improve FIB's performance on programs with high rates of serialized sharing. Figure 5 summarizes the full adaptive FIB ownership protocol. We motivate the adaptive extensions with examples demonstrating limitations of the core FIB ownership policy under extensive serialized sharing (§6.1).

6.1 Motivation: Tracking Serialized Sharing

FIB's pure derivation of ownership from access history is a critical strength that allows the protocol to deliver analysis atomicity on thread-local and read-shared accesses with zero time or space overhead relative to a non-atomic analysis. Yet the pure derivation is also a limitation. Some programs with highly contended serialized sharing cause frequent conflicting transitions, accruing time overhead that outweighs the savings of zero-cost atomicity in others. Strictly pure ownership derivation binds ownership transitions to analysis updates and precludes some desirable preemptive ownership transitions that could avoid expensive coordination at low cost, because such transitions must be effected by analysis updates that would invalidate the unified analysis state.

Consider the following example trace, which shows several threads accessing location x, serialized by lock l:

$$\mathsf{w}_{t_1}x;\ \mathsf{r}_{t_1}x;\ \mathsf{rel}_{t_1}l;\quad \mathsf{acq}_{t_3}l;\ \mathsf{r}_{t_3}x;\ \mathsf{rel}_{t_3}l;\quad \mathsf{acq}_{t_4}l;\ \mathsf{r}_{t_4}x;\ \mathsf{rel}_{t_4}l;\quad \mathsf{acq}_{t_2}l;\ \mathsf{r}_{t_2}x;\ \mathsf{rel}_{t_2}l$$

Since all accesses to x in this trace are totally ordered by happens-before, FTO and FIB will maintain a single last-read epoch for x. Location x will thus remain in an Excl state, taking a chain of expensive

single-conflict transition at each access following the read by thread t_1: $\text{Excl}(t_1) \rightarrow \text{Excl}(t_3) \rightarrow \text{Excl}(t_4) \rightarrow \text{Excl}(t_2)$.

There are two reasons why these transitions are unnecessary. First, although the reads are serialized, it is also reasonable to describe this as a read sharing pattern. Second, as every access to x in this trace is protected by lock l, analysis barriers will execute atomically by virtue of the program structure. Predictive read sharing (§6.2) and adaptive support for serialized write sharing (§6.3) exploit these properties to avoid expensive transitions in this and similar traces.

6.2 Optimization: Predictive Read Sharing

Predictive read sharing adaptively elects to use a *read map* representation of a location's last reads (§7) more frequently than the FastTrack or FastTrack-Ownership (FTO) algorithms, when doing so could likely eliminate expensive conflicting FIB transitions. For example, in the trace shown in §6.1, a preemptive transition to $\text{Shared}(\{t_1\})$ at thread t_1's first read of x has no coordination cost, since t_1 owns the location. This choice preserves soundness or completeness and would eliminate *all* conflicting transitions in the trace. Even delaying this transition to t_3's read would incur only one conflicting transition and save additional conflicting transitions going forward.

Predictive read sharing adaptively elects to represent past reads with a read map when serialized or concurrent reads (but no writes) are likely to follow in the near future. Mispredicting read sharing increases the cost of following thread-local writes, so we apply a simple heuristic to initiate read sharing in only some cases beyond where it is strictly necessary. Read sharing begins speculatively on a read by thread t when there is a single last read that happens before the current operation ($R_x \leq C_t$) and the last write is by a thread other than the current one ($W_x = c@u$ where $t \neq u$). Otherwise, the standard FTO policy applies. As in the standard protocol, read sharing terminates at the first subsequent write. Alternative termination policies are feasible, but not explored in this work. This is a rare opportunity in FIB ownership derivation where an adjustment in ownership is feasible while preserving valid analysis metadata.

In the example trace in §6.1, x would remain in state $\text{Excl}(t_1)$ at t_1's read of x. At thread t_3's read of x, a single-conflict transition would occur, but the predictive read sharing heuristic would make the transition to $\text{Shared}(\{t_1, t_3\})$ state. Following reads would augment the Shared state without conflicting transitions.

6.3 Optimization: Adapting to Serialized Write Sharing

Adaptively switching between the FIB ownership protocol and a conventional synchronization protocol allows the data race detector to exploit FIB's low costs on thread-local or read-shared data and shift to conventional synchronization to handle frequent serialized sharing that would cause many conflicting FIB transitions.

Consider the following example trace, modified from §6.1 to introduce more writes:

$\text{w}_{t_1}x;\ \text{r}_{t_1}x;\ \text{rel}_{t_1}l;\quad \text{acq}_{t_3}l;\ \text{r}_{t_3}x;\ \text{w}_{t_3}x;\ \text{rel}_{t_3}l;\quad \text{acq}_{t_4}l;\ \text{r}_{t_4}x;\ \text{w}_{t_4}x;\ \text{rel}_{t_4}l\quad \text{acq}_{t_2}l;\ \text{r}_{t_2}x;\ \text{rel}_{t_2}l;$

Under the pure FIB protocol, each new thread's access triggers a conflicting transition, as ownership is passed from thread to thread. Conflicting transitions occur at a rate approaching 50%. The predictive read sharing extension does not help with serialized *write* sharing. In this case it is reasonable to fall back on conventional synchronization for analysis, using a scheme like that described in §4.1.

Figure 5 shows the augmented FIB protocol extended with predictive read sharing and adaptive serial support. Under the adaptive policy, each access history is augmented with additional storage for a conflict counter and a bit indicating FIB vs. Serial mode. All access histories begin in FIB ownership mode. When an access history takes a conflicting transition, its conflict counter is
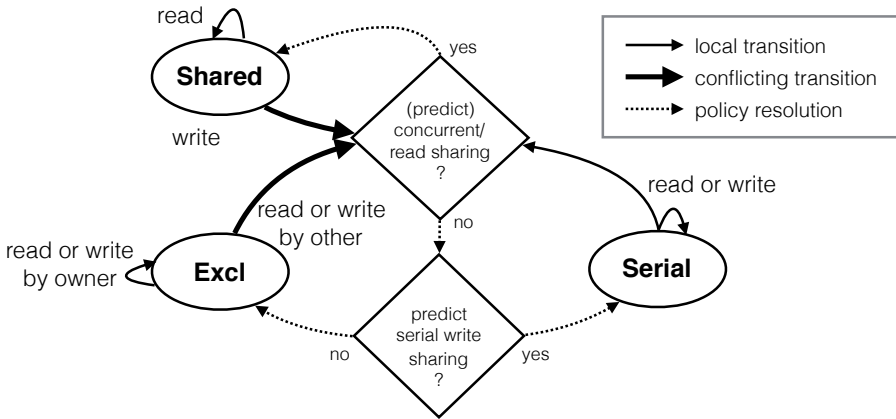
Fig. 5. FIB protocol with adaptive synchronization policies for serialized write-sharing.

incremented. When this counter reaches a threshold, the responding thread places the access history into Serial mode, allowing future transitions to occur locally with conventional optimistic synchronization.

When concurrent reads must be recorded in the access history, or when upcoming sharing is predicted, the access history returns to Shared FIB state to take advantage of zero-cost atomicity in that state. The current adaptive policy does not otherwise revert from Serial mode to FIB mode.

7 IMPLEMENTATION

We implemented several variations of the FastTrack-Ownership algorithm using FIB and other barrier atomicity protocols in the Jikes RVM Java virtual machine [Alpern et al. 1999] version 3.1.3.[3] This section describes key implementation choices and comparison with prior FastTrack implementations.

7.1 Common Analysis Metadata and Instrumentation

All FastTrack-Ownership configurations share the same representations of access history and the same synchronization tracking support. Configurations differ only in access barriers, yield points, and extended access history storage for the adaptive policies. Instrumentation is applied to all classes used by an application, including classes in the standard library.

7.1.1 Analysis Metadata. An access history, x, is encoded as two adjacent words in memory, the *last-read word* (x.R or R_x) and the *last-write word* (x.W or W_x). Access histories for object and static fields are inserted alongside object and static fields, respectively. A header word in each data array points to a lazily allocated shadow array of histories for data array elements.

The last-write word of an access history always stores an epoch. The last-read word stores an epoch or a pointer to a read map, distinguished by the least significant bit. The garbage collector scans last-read fields for read-map references and ignores epochs. Jikes RVM supports x86 only in 32-bit mode, so we represent epochs as 32 bits to simplify atomic loads and stores despite the limited encoding space. After the epoch tag bit, the next lowest 5 bits represent the thread identifier. The remaining 26 bits represent a logical clock. This allocation balances clock overflow and thread identifier exhaustion in the 32-bit space, but it limits this prototype to programs using no more

---

[3]http://www.jikesrvm.org

than 32 threads, counted by 5 bits, which in turn limits the scope of programs we can evaluate.[4] This limits our prototype only, not the FastTrack analysis or FIB protocol in general. A 64-bit JVM would more naturally support 64-bit epochs with space to count many more threads and clocks.

Vector clocks and read maps are fixed-length arrays of epochs, one per thread. Note that fixed-length arrays are enabled in part by the 32-thread limit. Supporting arbitrary large numbers of threads would make flexibly-sized read maps more appealing, likely by a "stop and copy" approach. Fortunately, the FIB protocol already provides the necessary cross-thread communication support that could be reused to safely coordinate replacement of a read map on the fly.

Each thread stores a vector clock and a separate copy of its current epoch to avoid extra indirection in same-epoch fast paths. Since every Java object is a monitor (lock), each object header contains a reference to a lazily allocated vector clock to track the lock's synchronization ordering. Vector clocks tracking synchronization on arrays are referenced by the header of the data array's shadow array to avoid adding a second header word. Java `volatile` fields are also shadowed by vector clocks. Classes are shadowed by a vector clock representing their logical time of initialization.

7.1.2 Instrumentation. Thread fork and join, lock acquire and release (including via `wait`), and `volatile` field load and store operations are treated via the standard vector-clock algorithm to capture happens-before ordering induced under the Java Memory Model [Flanagan and Freund 2009; Manson et al. 2005]. Most synchronization tracking is skipped until the main thread first forks a new thread, since synchronization order induced during this single-threaded execution prefix is redundant with program order in the main (and only) thread.

The Java Memory Model also relates static field accesses in a class to the class initialization by happens-before [Biswas et al. 2015; Manson et al. 2005]. To track this ordering, a vector clock shadowing the class records the current logical time when class initialization completes. Before each static access, the vector clock of the relevant class is merged into the current thread's vector clock, representing acquire-style synchronization. To avoid redundant expensive merges of immutable class initialization vector clocks on repeated static accesses, each thread keeps a bit vector of the classes whose initialization it has already observed and performs the merge only for non-observed classes. As with thread vector clocks, a new child thread inherits a copy of its parent thread's class initialization bit vector when started. The prototype does not currently propagate these large bit vectors across other more frequent synchronization edges.

Access analysis fast paths, including standard same-epoch fast paths plus much of the barriers in Figure 4, are inlined before accesses. Inlined fast paths call out-of-line slow paths for coordination and other rare events as needed. FIB uses existing yield points in Jikes RVM to manage communication. All implementations report the first data race on each location. Since the FIB protocol implementation is optimized under the assumption that data race freedom is the common case, the case of a detected race comes with high cost. Since all algorithms are sound only to the first data race on a location and some programs exhibit a non-trivial number of data races, we configure all data race detectors to end tracking on a location after its first data race.

7.2 Data Race Detectors

We implemented several variations on FastTrack-Ownership (FTO) with different implementations of analysis atomicity in Jikes RVM:

- FTOUNSYNC uses FTO optimizations assuming analysis atomicity, yet it makes no effort to ensure analysis atomicity, making it inaccurate. It serves as an approximate ground truth for ideal performance of atomic analysis barriers.

---

[4]The choice of 5 thread ID bits forces the 32-thread limit. The correspondence of 32 bits and 32 threads is coincidental.

- FTOlock provides atomic FTO barriers with naïve use of lightweight spin locks on the last-read word of metadata.
- FTOcas implements the scheme shown in Figure 2 and discussed in §4.1, using CAS only when updates are required.
- FTOfib implements the base Fib protocol (§5).
- FTOfib+share extends FTOfib with predictive read sharing (§6.2).
- FTOfib+adapt extends FTOfib with adaptive synchronization for write sharing (§6.3).
- FTOfib+adapt(so)+share combines predictive read sharing with the adaptive write-sharing optimization for static fields only, with heuristics tuned to start static fields in Serial mode.

To evaluate the performance of FTO versus the original FastTrack algorithm, we also implement FTcas, a close reproduction of the original algorithm in [Flanagan and Freund 2009], with the addition of the read shared same epoch case as in [Biswas et al. 2015; Flanagan and Freund 2017]. FTcas reuses all infrastructure described in §7.1. FTcas analysis cases use a CAS or light spin lock when updating the access history. FTcas inlines only same-epoch cases. FTOfib and other FTO configurations also inline their additional synchronization-free cases.

7.3 Fib Infrastructure

A thread's incoming request and ack queues are represented as a pair of one-word bit vectors in the thread's structure. The queues are unordered since co-resident requests are inherently concurrent. Each thread receives at most one request at a time from any other thread, so queue size is bounded by the maximum number of threads (32). Request and response arguments, including target location, type of access, and result, are stored in the requesting thread's structure. Ack requests and responses have no arguments. A thread's main request queue is the central synchronization point for operations that must execute atomically with respect to the thread's owned access histories.

A requesting thread CASes out the responding thread's bit vector, replacing it with a special *locked* value. It then inspects the last-read word of the access history it is requesting and if it is still owned by the same thread, it writes back a new bit vector with the requesting thread's bit set. When a thread blocks after a yield point it CASes its queue to a special *blocked* value. Each thread's communication state is visible through this single word, simplifying atomic transitions.

Responses are stored in the requesting thread's structure. After sending a request, the requesting thread awaits a response placed here by the requesting thread. While awaiting responses, requesting threads block or periodically issue yield points to process their own incoming request queues to avoid deadlock. Ack responses accumulate a bit vector or counter of responding threads.

Adaptive synchronization support (FTOfib+adapt, FTOfib+adapt(so)+share) steals an additional bit from epoch clocks to indicate ownership mode (pure Fib or Serial) and adds two words to each access history to store a conflict counter. A single word (or less) would suffice, but our prototype currently supports only power-of-2-sized access history sizes.

7.4 Comparison with ValorFT Implementation

For comparison, we examine the FastTrack implementation from [Biswas et al. 2015], which we call ValorFT. It is the closest prior FastTrack implementation, also based in Jikes RVM. ValorFT uses similar instrumentation strategies for synchronization and some shared compiler support for inserting access analysis barriers. Overall it is closest to FTcas. Nonetheless, the implementation differences discussed below impact performance, evaluated in §8.4.

7.4.1 Access History Representation. The ValorFT implementation inlines access histories in object layouts in the same style as the other configurations in this work. While the other implementations represent each epoch in one 32-bit word, ValorFT represents each epoch as a pair

of 32-bit words. VALORFT thus requires twice the loads or stores to effect access history operations. On the other hand, VALORFT's two-word epochs are free from caps on thread counts and clock sizes that arise from 32-bit epochs. Single-word 64-bit epochs in a 64-bit JVM would be preferable.

Both implementations store read maps as separate objects in the heap. VALORFT uses specialized hash tables that are space efficient for sparse read maps but involve multiple indirections to access individual entries. In contrast, our implementations use fixed-size arrays that are space inefficient for sparse read maps, but support map entry access via a single memory operation. VALORFT's read map representation requires modest extra synchronization to protect read map hash table growth.

7.4.2 Analysis Barriers: Inlining, Atomicity, and Reporting. The VALORFT implementation uses analysis logic, inlining, and an atomicity scheme similar to FTCAS, but requires minor additional synchronization for access history manipulations due to the access history representation (§7.4.1). Extra memory operations to load and store larger epochs also increase the length of analysis critical sections, although contention – indicative of a data race or the initiation of read sharing – should occur rarely in well-synchronized programs. Our implementations report only the first data race on a location (§7.1.2). VALORFT continues best-effort detection past the first data race on a location.

7.4.3 Class Initialization Tracking. The VALORFT implementation does not employ the class initialization memoization optimization described in §7.1.2, instead performing an expensive vector clock merge on every static access.

8 EVALUATION

This section evaluates the performance of several data race detection implementations to characterize the performance impact of analysis atomicity via FIB, its adaptive extensions, and other approaches. The experimental evaluation aims to answer three key questions:

(1) How closely does FIB approach ideal zero-cost analysis atomicity?
(2) How does FIB's performance compare to approaches using conventional synchronization?
(3) How well does FIB's model of common local transitions and uncommon conflicting transitions match real application behavior?

This section describes experimental methodology (§8.1) and reports results of performance experiments to measure the impact of FIB, its adaptive extensions, and other approaches on running time and space (§8.2). Profiling results in (§8.3) help characterize empirical behavior of the FIB protocol and its performance impact. Additionally, §8.4 characterizes the overall performance of our data race detection implementations in the context of prior implementations.

8.1 Methodology

8.1.1 Evaluated Programs. We ran performance and profiling experiments on multithreaded Java programs from the DaCapo benchmark suites [Blackburn et al. 2006] versions 2006-10-MR2 (eclipse6 and xalan6) and 9.12 (avrora9, jython9, luindex9, lusearch9 (fixed per [Yang et al. 2011]), pmd9, sunflow9, and xalan9) using large workloads, plus pjbb2005,[5] a fixed-workload version of the SPECjbb2005 benchmark [Standard Performance Evaluation Corporation 2005]. We omit DaCapo benchmarks that exceed the thread limit of our prototype (§7.1.1), lack multithreading, or do not execute on the base Jikes RVM.

8.1.2 Experiments. We ran two groups of performance experiments with available cores capped at 8 and 30, respectively. We measured execution time and heap size for each. Programs internally choose a fixed or cores-dependent number of threads. A cap of 30 cores ensures that programs using

---

[5]http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005

n worker threads plus a main thread fit the 32-thread limit of our prototype (§7.1.1). Experiments for sunflow9 used 15 cores to fit the 32-thread cap, since sunflow9 creates 2 worker threads per core ($2 \times 15 + 1 = 31$). While the prototype's thread cap limits the scale of threading in evaluation, our experimental platform provides enough cores to execute all supported threads concurrently. We ran the FTO implementations described in §7.2, plus Base (Jikes RVM without data race detection), FTcas (§7.2) and ValorFT (§7.4). All implementations used Jikes RVM's FastAdaptive just-in-time compiler and GenImmix garbage collector, which we allow to adjust the heap size automatically. We ran separate profiling experiments on 30 cores, inserting additional counters in our FTO implementations. Each experimental configuration executed 10 times. Unless otherwise noted, discussion in following sections refers to results from 30-core configurations.

8.1.3   Platform. All experiments were executed on a machine with 2×18-core Intel Xeon E5-2695 v4 CPUs with a total of 256GB of RAM across two NUMA nodes. We selected the CPUs' performance mode and disabled simultaneous multithreading. The machine runs the Ubuntu 16.04 LTS distribution with Linux kernel version 4.4.0.

8.2   Performance Impact of Analysis Atomicity

Figure 6 plots results of our performance experiments, with running times of each data race detector normalized to the running time of Base (no data race detection), for 8 and 30 cores. sunflow9 15-core (30 worker threads plus main thread) results are shown with the 30-core results.

8.2.1   Execution Time Overhead of Conventional Synchronization. Using the running time of the completely unsynchronized FTOunsync configuration as an approximate ground truth for expected ideal synchronization performance, it is clear that the cost of atomic analysis barriers is high with trivial implementations, but reasonable in many programs using more sophisticated implementations. The suboptimal FTOlock implementation runs roughly 107% slower than FTOunsync on average across all programs, ranging from 15% on pjbb2005 to over 16 times slower on sunflow9. sunflow9's repeated concurrent read sharing is serialized by FTOlock's pessimistic access history locking, to the detriment of performance.

FTOcas achieves analysis atomicity with average overheads of just 6% over FTOunsync. These lower overheads demonstrate the extent to which even the simple FastTrack pure fast paths avoid synchronization overhead, including on sunflow9, where FTOcas has performance effectively equivalent to that of FTOunsync. There is little room for improvement overall beyond FTOcas without changing the analysis algorithm itself. Nonetheless, notable exceptions remain: FTOcas runs 32% slower than FTOunsync on xalan6 and 19% slower on eclipse6.

8.2.2   Execution Time Overhead of Fib. Across all programs, FTOfib's overhead relative to FTOunsync is about 14%, which is slower than FTOcas. However, FTOfib is competitive with FTOunsync on eclipse6, jython9, luindex9, lusearch9, and pmd9, and adaptive extensions improve Fib performance. FTOfib+share is less than 4% slower than FTOunsync or 1-2% faster than FTOcas.

Predictive read sharing indeed helps to avoid overheads incurred by serialized read sharing that would otherwise trigger expensive single-conflict transitions in Fib. Predictive read sharing makes a marked improvement for xalan6, avrora9, xalan9, and pjbb2005, where FTOfib runs 23-36% slower than FTOfib+share. FTOfib+share significantly reduces the overhead of FTOfib in these programs, making its performance competitive with FTOunsync.

Compared to FTOcas, FTOfib+share is competitive on average, but notably faster on specific programs. On xalan6, FTOcas runs 20% slower than FTOfib+share, while FTOfib+share is competitive with FTOunsync. FTOcas never outperforms the faster of FTOfib or FTOfib+share by
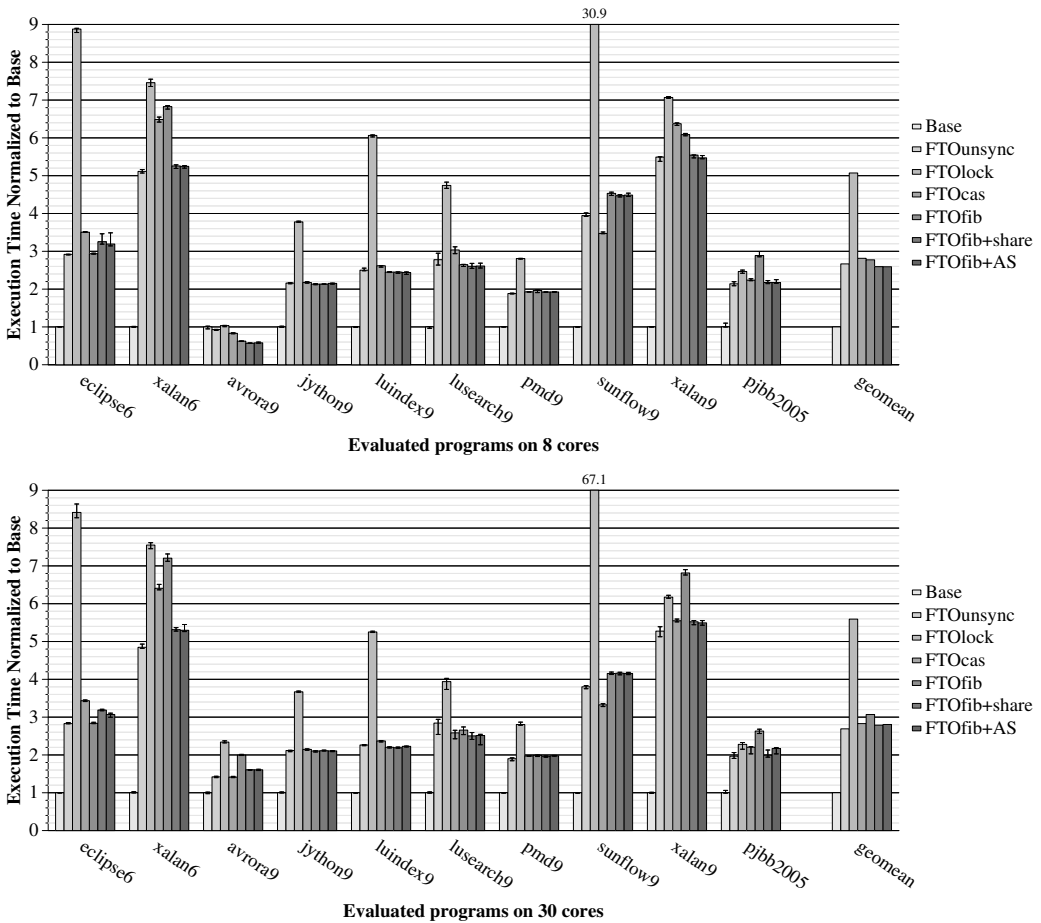
Fig. 6. Execution times of data race detector implementations normalized to Base Jikes RVM, with 95% confidence intervals. FTfib+AS abbreviates FTOfib+adapt(so)+share.

more than 20%, on sunflow9, where FTOcas also runs 13% faster than FTOunsync. Variability in execution time for FTOfib+share gains an advantage on 8 cores, where it is the fastest atomic analysis implementation, at about 8% faster than FTOcas on average, and competitive with FTOunsync.

The adaptive FTOfib+adapt configuration (not shown) is slower than FTOfib, averaging 27% slower than FTOunsync. FTOfib+adapt(so)+share, a combination of general predictive read sharing and adaptive Serial mode for static fields, achieves average overhead of 4%, similar to FTOfib+share and FTOcas. Further exploration of adaptive heuristics is necessary to determine if this extension can prove useful in combination with predictive read sharing.

8.2.3 Space Overhead. Figure 7 shows memory usage measurements for several configurations, normalized to the memory usage of Base Jikes RVM. All configurations implemented in this paper have comparable overheads between 130% and 145%. There is little variation between configurations. The main notable result is that predictive read sharing in FTOfib+share and FTOfib+adapt(so)+share introduces only modest additional overhead of well under 10% relative to FTOfib, despite introducing large read maps where they are not strictly necessary. Only pjbb2005
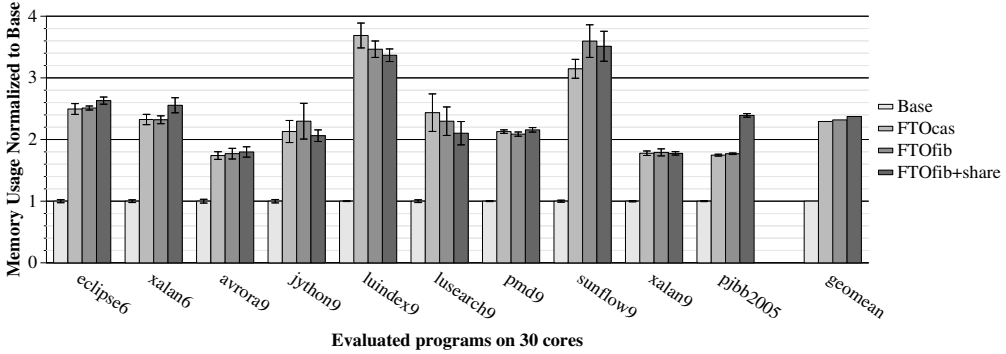
Fig. 7. Memory usage of data race detection implementations normalized to Base Jikes RVM, with 95% confidence intervals.

Table 4. Frequency of Fib ownership state transitions in FTOfib, FTOfib+adapt, and FTOfib+share.

		Pure Local	Local	Fence	First	Serial	Single-Conflict Excl → Excl	Excl → Shared	Multi-Conflict
eclipse6	FTOFIB	82.7%	9.72%	0.000443%	7.6%	0%	0.00475%	0.000143%	0.000096%
	FTOFIB+A	87.1%	5.28%	0.000304%	7.6%	0.011%	0.00489%	0.0000695%	0.0000186%
	FTOFIB+S	82.7%	9.73%	0.00252%	7.6%	0%	0.00262%	0.00164%	0.000272%
xalan6	FTOFIB	48.2%	45.8%	0.000166%	5.26%	0%	0.767%	0.00000771%	< 0.000001%
	FTOFIB+A	50.5%	42.2%	0.000161%	5.27%	2.01%	0.000791%	0.00000381%	< 0.000001%
	FTOFIB+S	48.1%	46.4%	0.000369%	5.26%	0%	0.175%	0.0000419%	0.0824%
avrora9	FTOFIB	92.2%	6.82%	0.0109%	0.573%	0%	0.343%	0.0000467%	< 0.000001%
	FTOFIB+A	92.2%	6.77%	0.011%	0.573%	0.378%	0.0343%	0.0000129%	< 0.000001%
	FTOFIB+S	92.2%	7.04%	0.0814%	0.573%	0%	0.068%	0.0075%	0.0137%
jython9	FTOFIB	88.8%	0.749%	0%	10.5%	0%	0%	0%	0%
	FTOFIB+A	88.8%	0.746%	0%	10.5%	0%	0%	0%	0%
	FTOFIB+S	88.8%	0.748%	0%	10.5%	0%	0%	0%	0%
luindex9	FTOFIB	93.8%	3.42%	0%	2.79%	0%	0.000291%	< 0.000001%	< 0.000001%
	FTOFIB+A	93.8%	3.42%	0%	2.79%	0%	0.00029%	< 0.000001%	< 0.000001%
	FTOFIB+S	93.8%	3.42%	0.00000216%	2.79%	0%	0.00029%	0.0000627%	0.00000192%
lusearch9	FTOFIB	84.9%	11.9%	0.00016%	3.14%	0%	0.0338%	0.00000203%	0%
	FTOFIB+A	84.9%	11.9%	0.000163%	3.14%	0.0359%	0.000365%	0.00000216%	0%
	FTOFIB+S	84.9%	11.9%	0.000336%	3.14%	0%	0.000316%	0.0000471%	0.00000999%
pmd9	FTOFIB	84.3%	7.02%	0.00516%	8.44%	0%	0.2%	0.000081%	< 0.000001%
	FTOFIB+A	84.3%	6.66%	0.00519%	8.45%	0.549%	0.0163%	0.0000726%	0.0000014%
	FTOFIB+S	84.3%	7.18%	0.0213%	8.44%	0%	0.0188%	0.00287%	0.00456%
xalan9	FTOFIB	54.0%	37.9%	0.00128%	7.52%	0%	0.617%	0.0000577%	< 0.000001%
	FTOFIB+A	54.0%	37.4%	0.00122%	7.52%	1.13%	0.00125%	0.0000152%	< 0.000001%
	FTOFIB+S	54.0%	38.2%	0.00264%	7.51%	0%	0.211%	0.0000784%	0.0906%
pjbb2005	FTOFIB	51.3%	39.9%	0.000385%	8.09%	0%	0.716%	0.0000197%	< 0.000001%
	FTOFIB+A	51.3%	39.4%	0.000412%	8.1%	1.19%	0.0628%	0.0000426%	< 0.000001%
	FTOFIB+S	51.2%	40.6%	0.0214%	8.12%	0%	0.0457%	0.00481%	0.000631%

incurs significant overhead for predictive read sharing (§8.2.2). Execution time improvements from this optimization come at modest space cost in all other programs in practice.

8.3　Fib Ownership Transition Characterization

Table 4 shows results from separate profiling experiments run on 30 cores (except sunflow9, on 15 cores) to measure the frequency of ownership state transitions on accesses in the Fib protocol and
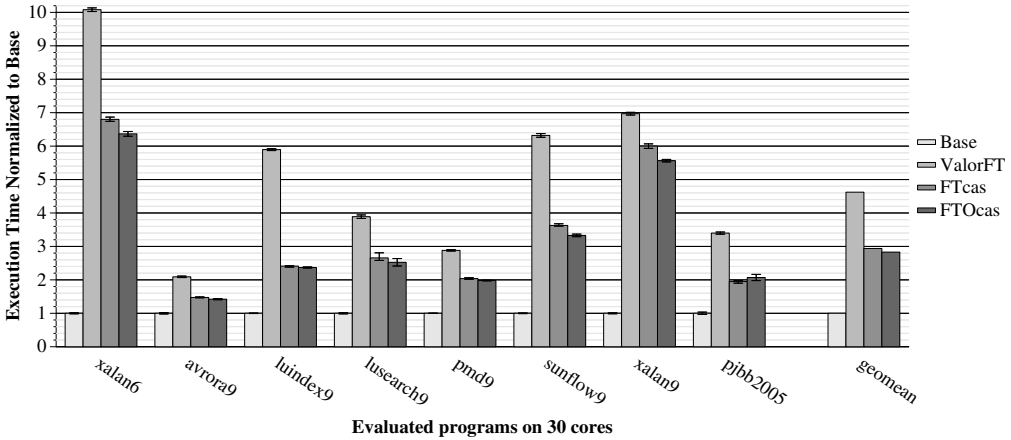
Fig. 8. Execution time of VALORFT, FTCAS, and FTOCAS normalized to BASE Jikes RVM, on selected programs, with 95% confidence intervals.

its adaptive extensions. Groups of rows in this table show results for each configuration on each program. Columns indicate the type of ownership state transition, as discussed in §5 (local, fence, single conflict, multiple conflict) and §6.3 (serial). The column labeled *First* indicates initial accesses to fresh locations, handled in our implementation by a single CAS that is typically uncontended. Columns are grouped into pure transitions and transitions that make access history updates. Each cell lists the frequency of its transition in its configuration as a percentage of all accesses.

Profiling results in Table 4 show that in programs where FIB performs well, rates of expensive conflicting transitions are typically well under 0.1% and often significantly lower, justifying FIB's approach of making these rare operations responsible for a large majority of the coordination effort to maintain atomicity. Zero-cost local transitions and lightweight first-access transitions account for an overwhelming majority of all accesses.

These results also show that predictive read sharing often reduces the rate of expensive conflicting FIB transitions by an order of magnitude or more in FTOFIB+SHARE. The optimization is particularly effective in improving performance for xalan6, avrora9, xalan9, and pjbb2005 because it reduces their high rates of expensive conflicting FIB ownership transitions (around 0.7%) by about three times or more. While memory allocation and the number of multiple-conflict transitions grow modestly as a result of more frequent use of last read maps, the number of these transitions is small in absolute and relative terms, compared to the overall reduction in conflicting transitions.

8.4 Performance Comparison with Prior Work

This paper focuses on reducing overheads of analysis atomicity relative to the base cost of data race detection analysis logic. Nonetheless, the overall performance cost of the data race detection implementations presented here is lower than – though not directly comparable with – costs reported for some prior implementations, including an implementation in Jikes RVM [Biswas et al. 2015] and implementations based on the bytecode-instrumenting RoadRunner analysis framework [Flanagan and Freund 2009, 2010; Rhodes et al. 2017]. To ground the performance of FastTrack-Ownership and FIB relative to prior work, we also evaluate the standard FastTrack algorithm ([Flanagan and Freund 2009], §2) both as implemented using our own infrastructure (FTCAS, §7.2) and in a prior independent implementation (VALORFT [Biswas et al. 2015], §7.4).
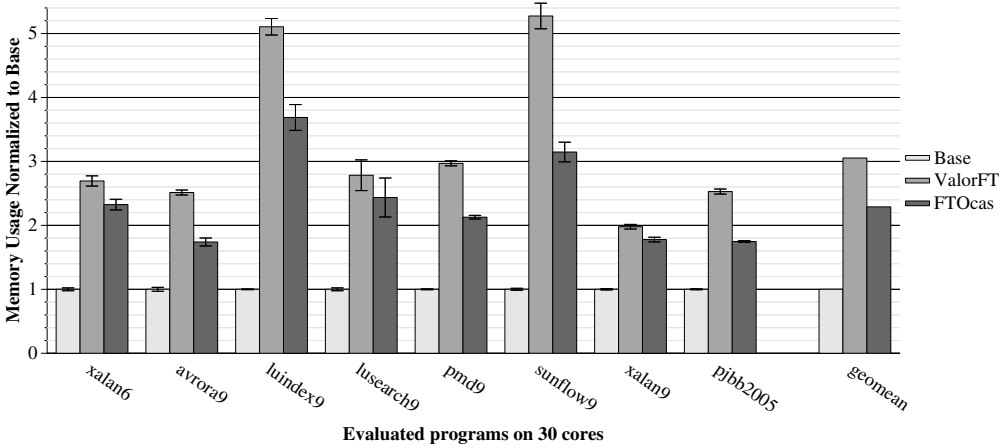
Fig. 9. Memory usage of VALORFT and FTOCAS normalized to BASE Jikes RVM, on selected programs, with 95% confidence intervals.

Figure 8 shows results of 30-core experiments (§8.1.3). The plot shows execution times of VALORFT, FTCAS, and FTOCAS, normalized to the execution time of BASE Jikes RVM. Figure 9 shows memory usage of multiple configurations normalized to memory usage of BASE Jikes RVM. Figures 8 and 9 omit eclipse6 and jython9 due to issues with 32-bit virtual memory space and running time of VALORFT on our evaluation platform. By profiling analysis events in VALORFT, FTCAS, and FTOCAS, the three closest implementations, we confirmed that they analyze the same accesses and synchronization events in practice, within margins of variability between executions.

On average, across all evaluated programs (including eclipse6 and jython9, not shown in Figure 8), FTOCAS runs 3% faster than the original FTCAS. By sharing all infrastructure except analysis barriers, this comparison isolates the costs of FastTrack and FTO, indicating that they are competitive, with FTO showing a small improvement overall.

VALORFT runs 57% slower than FTCAS and 64% slower than FTOCAS on average, across the programs shown in Figure 8. Based on profiling and study of the VALORFT code base, these differences in performance appear to arise from the key implementation differences outlined in §7.4.

The use of two-word epochs in VALORFT may grow the cost of analysis barriers by doubling the number of memory operations in many cases. It also grows the memory footprint 32% over FTOCAS, as shown in Figure 9. This margin also reflects space savings in VALORFT from its more flexible read maps. However, VALORFT's hash table read maps make read map operations more costly in time than direct array indexing in the other implementations. For example, VALORFT has relatively high overhead on sunflow9, which involves extensive read sharing.

VALORFT and FTCAS dispatch fewer accesses with inlined analysis cases than do FTO configurations. However, the small performance difference between FTCAS, which uses lightweight inlining similar to VALORFT, and FTOCAS, which uses more aggressive inlining, suggests that different levels of inlining likely do not contribute significantly to VALORFT's larger overheads.

Tracking ordering from class initialization to static field access (§7.1.2, §7.4.3) appears to have nontrivial impact on performance difference. VALORFT performs a vector clock merge operation on every static access, affecting up to 8% of all accesses of any kind in some evaluated programs. Vector clock merge operations, which have running time linear in the number of threads, are substantially more expensive than any common case read or write analysis barrier. In contrast, optimizations

in FTCAS and all FTO implementations require vector clock operations to track class initialization ordering on at most 0.000001% of all accesses.

9 RELATED WORK

Relevant prior work on data race detection is reviewed in §1.1 and §2. Cooperative synchronization based on thread ownership has been employed in a range of settings previously. The most relevant to FIB are biased locking, cache coherence and associated techniques like RADISH local permissions, and object-granularity conflict detection systems.

9.1 Biased Locking

Biased locking [Kawachiya et al. 2002; Pizlo et al. 2011; Russell and Detlefs 2006; Vasudevan et al. 2010] is a well-known implementation technique suited to locks that are acquired by only one thread, at least within some large consecutive series of acquires. Initially the lock is unowned and must be acquired in the typical fashion. A thread may bias a lock it acquires with its thread ID. Once a lock is biased, the bias owner may acquire it and release it without fences, CAS, or other synchronization. If another thread tries to acquire the lock, it must request unbiasing or a bias transfer from the bias owner. Threads check for requests at yield points. Inter-thread requests are more expensive than standard lock operations, but also much rarer.

FIB goes further than biased locking of access histories by deriving bias directly from access histories and subsuming bias checks in analysis work. Pure FIB has no unbiased mode; the adaptive Serial mode is analogous to an unbiased mode.

9.2 Coherence and Permissions

Cache coherence protocols such as MESI [Papamarcos and Patel 1984] ensure data duplicated in private caches on separate cores do not diverge. When a core has a cache line in sufficiently permissive ownership state, it can access the line without communication. Otherwise it must communicate to acquire ownership and ensure up-to-date data. The RADISH [Devietti et al. 2012] software–hardware data race detector uses cache coherence events to optimize its analysis. A data-race-free check result on location x applies to all subsequent accesses to x up to the next cache coherence or eviction on the same cache line. FIB uses a similar ownership protocol, without requiring hardware support, and without duplicating access histories or memoizing checks.

9.3 Object Race Detection and Octet

Per-object thread ownership has been used as an optimization to filter out data race checks in object race detection [von Praun and Gross 2001], but object-granularity race detection is imprecise. FIB neither misses true data races nor reports false data races.

Octet [Bond et al. 2013] associates an ownership state with each object to report conflicting state transitions to client analyses. As an object-granularity system, Octet is not suited to perform accurate fine-grained data race detection. FIB's ownership system and cooperative communication protocol is similar to Octet, but operates at the granularity of individual fields and derives ownership state purely from existing metadata rather than requiring explicit additional storage. Object granularity can reduce the number of expensive transitions required if many fields of an object experience identical ownership transitions, but false sharing may also induce the opposite effect. To support accurate data race detection with object ownership, ownership storage and checks must be decoupled from analysis metadata and logic, introducing costs FIB does not have.

An extension of Octet adaptively selects between cooperative Octet communication and competitive CAS-based synchronization to address mismatch between costs and rates of common or uncommon cases in the base Octet protocol [Cao et al. 2016]. Again, FIB's adaptive ownership

policies are distinguished from adaptive Octet by granularity and the degree to which they integrate with existing metadata or synchronization. A sophisticated cost model and adaptive heuristics are important to manage coordination at a coarser object granularity in Octet. In contrast, FIB must handle field granularity, but currently employs only simple adaptive heuristics.

A key strength of the pure FIB protocol, relative to Octet and other cooperative ownership systems described above, is the ability to provide atomicity guarantees with zero additional work in common cases by exploiting work that is already being done by the data race detection logic.

9.4 Eliminating Redundant Analysis Barriers

Researchers have developed several techniques to reduce the cost of data race detection or related analyses by eliminating analysis checks or barriers that are guaranteed to be redundant with other analysis barriers due either to repeated accesses to the same location (due to temporal locality) or related locations (including due to spatial locality) during a single epoch [Bond et al. 2013; Flanagan and Freund 2013; Peng and Devietti 2015; Rhodes et al. 2017; Wilcox et al. 2015]. While FIB does not eliminate such redundant barriers directly, it naturally ensures that redundant analysis barriers on the same access history are inexpensive, dispatched locally by pure synchronization-free cases. Techniques for redundancy elimination are complementary to FIB.

10 CONCLUSIONS

FIB is a novel cooperative synchronization protocol designed to enforce analysis atomicity for dynamic data race detection metadata with zero added cost on common-case operations and cross-thread coordination in uncommon cases. FIB derives ownership state for a location's access history purely from the preexisting data race detection access history metadata itself. Checking ownership state is subsumed by preexisting data race detection analysis logic. Ownership states grant analysis permissions. State transitions require coordination among threads. Simple adaptive optimizations eliminate overheads in programs that otherwise induce frequent expensive state transitions. Our evaluation shows that FIB outperforms simple or pessimistic analysis synchronization schemes and is competitive with a tuned optimistic scheme. The FIB protocol affords new perspectives on the structure of data race detection and data-race-free program execution.

REFERENCES

Sarita Adve. 2010. Data races are evil with no exceptions. *Commun. ACM* 53, 11 (Nov. 2010), 84.

Sarita V. Adve and Hans-Juergen Boehm. 2010. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Commun. ACM* 53 (Aug. 2010). Issue 8.

Sarita V. Adve and K. Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29, 12 (1996), 66–76.

Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering—A New Definition. In *ACM/IEEE International Symposium on Computer Architecture*.

Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. 1991. Detecting Data Races on Weak Memory Systems. In *ACM/IEEE International Symposium on Computer Architecture*.

Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen E. Smith. 1999. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. http://www.jikesrvm.org.

David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. 1998. Thin Locks: Featherweight Synchronization for Java. In *ACM Conference on Programming Language Design and Implementation*.

Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. 2006. A Theory of Data Race Detection. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*.

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, Software-Only Region Conflict Exceptions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Stephen M. Blackburn, Robin Garner, Chris Hoffman, Asiad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *ACM Conference on Programming Language Design and Implementation*.

Hans-Juergen Boehm and Sarita V. Adve. 2012. You Don't Know Jack About Shared Variables or Memory Models. *Commun. ACM* 55, 2 (Feb. 2012), 48–54.

Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. 2013. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Man Cao, Minjia Zhang, Aritra Sengupta, and Michael D. Bond. 2016. Drinking from Both Glasses: Combining Pessimistic and Optimistic Tracking of Cross-thread Dependences. In *ACM Symposium on Principles and Practice of Parallel Programming*.

Luis Ceze, Joseph Devietti, Brandon Lucia, and Shaz Qadeer. 2009. A Case for System Support for Concurrency Exceptions. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*.

Jong-Deok Choi and Sang Lyul Min. 1991. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *ACM Symposium on Principles and Practice of Parallel Programming*.

Mark Christaens and Koen De Bosschere. 2001. A Topological Approach to On-the-fly Race Detection in Java Programs. In *Symposium on Java Virtual Machine Research and Technology*.

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. 2012. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ACM/IEEE International Symposium on Computer Architecture*.

Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-Juergen Boehm. 2012. IFRit: Interference-free Regions for Dynamic Data-Race Detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-Aware Java Runtime. In *ACM Conference on Programming Language Design and Implementation*.

Colin Fidge. 1991. Logical Time in Distributed Computing Systems. *IEEE Computer* 24 (August 1991). Issue 8.

Stephen J. Fink and Feng Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*.

Cormac Flanagan and Stephen N. Freund. 2000. Type-Based Race Detection for Java. In *ACM Conference on Programming Language Design and Implementation*.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *ACM Conference on Programming Language Design and Implementation*.

Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*.

Cormac Flanagan and Stephen N. Freund. 2013. RedCard: Redundant Check Elimination For Dynamic Race Detectors. In *European Conference on Object-Oriented Programming*.

Cormac Flanagan and Stephen N. Freund. 2017. *The FastTrack2 Race Detector*. Technical Report CSTR201701. Williams College. Working draft – accessed 25 August 2017.

Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound And Complete Dynamic Atomicity Checker for Multithreaded Programs. In *ACM Conference on Programming Language Design and Implementation*.

K. Kawachiya, A. Koseki, and T. Onodera. 2002. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21 (July 1978). Issue 7.

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691.

Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-Juergen Boehm. 2010. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ACM/IEEE International Symposium on Computer Architecture*.

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *ACM Symposium on Principles of Programming Languages*.

Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *ACM Conference on Programming Language Design and Implementation*.

Hassan Salehe Matar, Ismail Kuru, Serdar Tasiran, and Roman Dementiev. 2014. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *Workshop on Determinism and Correctness in Parallel Programming*.

Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *International Workshop on Parallel and Distributed Algorithms*. 215–226.

Sang L. Min and Jong-Deok Choi. 1991. An Efficient Cache-based Access Anomaly Detection Scheme. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. 2009. SigRace: Signature-Based Data Race Detection. In *ACM/IEEE International Symposium on Computer Architecture*.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *ACM Conference on Programming Language Design and Implementation*.

Takuya Nakaike and Maged M. Michael. 2010. Lock Elision for Read-Only Critical Sections in Java. In *ACM Conference on Programming Language Design and Implementation*.

Robert H. B. Netzer and Barton P. Miller. 1991. Improving the Accuracy of Data Race Detection. In *ACM Symposium on Principles and Practice of Parallel Programming*.

Robert H. B. Netzer and Barton P. Miller. 1992. What Are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems* 1, 1 (March 1992), 74âĂŞ88.

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

Mark S. Papamarcos and Janak H. Patel. 1984. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ACM/IEEE International Symposium on Computer Architecture*.

Yuanfeng Peng and Joseph Devietti. 2015. SlimFast: Reducing Metadata Redundancy in Sound & Complete Dynamic Data Race Detection. In *PLDI Student Research Competition*.

Yuanfeng Peng, Benjamin P. Wood, and Joseph Devietti. 2017. PARSNIP: Performant Architecture for Race Safety with No Impact on Precision. In *ACM/IEEE International Symposium on Microarchitecture*.

Filip Pizlo, Daniel Frampton, and Antony L. Hosking. 2011. Fine-grained Adaptive Biased Locking. In *International Conference on Principles and Practice of Programming in Java*.

Milos Prvulovic. 2006. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *International Symposium on High-Performance Computer Architecture*.

Milos Prvulovic and Josep Torrellas. 2003. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *ACM/IEEE International Symposium on Computer Architecture*.

Ravi Rajwar and James R. Goodman. 2001. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *ACM/IEEE International Symposium on Microarchitecture*.

Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. BigFoot: Static Check Placement for Dynamic Race Detection. In *ACM Conference on Programming Language Design and Implementation*.

Ian Rogers and Balaji Iyengar. 2011. Reducing Biased Lock Revocation By Learning. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*.

K. Russell and D. Detlefs. 2006. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* 15, 4 (1997).

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications*.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97.

Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. 2007. Enforcing Isolation and Ordering in STM. In *ACM Conference on Programming Language Design and Implementation*.

Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd D. Millstein, and Madanlal Musuvathi. 2011. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

Standard Performance Evaluation Corporation. 2005. SPECjbb2005. http://www.spec.org/jbb2005/. (2005).

Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. 2010. Simple and Fast Biased Locks. In *International Conference on Parallel Architectures and Compilation Techniques*.

Christoph von Praun and Thomas Gross. 2001. Object Race Detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. 2015. Array Shadow State Compression for Precise Dynamic Race Detection. In *IEEE/ACM International Conference on Automated Software Engineering*.

Benjamin P. Wood, Luis Ceze, and Dan Grossman. 2014. Low-Level Detection of Language-Level Data Races with LARD. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. 2011. Why Nothing Matters: The Impact of Zeroing. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *ACM Symposium on Operating Systems Principles*.

P. Zhou, R. Teodorescu, and Y. Zhou. 2007. HARD: Hardware-Assisted Lockset-based Race Detection. In *International Symposium on High-Performance Computer Architecture*.