RADISH: Always-On Sound and Complete Race Detection in Software and Hardware

Joseph Devietti❦, Benjamin P. Wood❦, Karin Strauss*, Luis Ceze❦, Dan Grossman❦, Shaz Qadeer*
❦University of Washington, *Microsoft Research
{devietti,bpw,luisceze,djg}@cs.washington.edu, {kstrauss,qadeer}@microsoft.com

Abstract

Data-race freedom is a valuable safety property for multithreaded programs that helps with catching bugs, simplifying memory consistency model semantics, and verifying and enforcing both atomicity and determinism. Unfortunately, existing software-only dynamic race detectors are precise but slow; proposals with hardware support offer higher performance but are imprecise. Both precision and performance are necessary to achieve the many advantages always-on dynamic race detection could provide.

To resolve this trade-off, we propose RADISH*, a hybrid hardware-software dynamic race detector that is* always-on and fully precise. *In* RADISH*, hardware caches a principled subset of the metadata necessary for race detection; this subset allows the vast majority of race checks to occur completely in hardware. A flexible software layer handles persistence of race detection metadata on cache evictions and occasional queries to this expanded set of metadata. We show that* RADISH *is correct by proving equivalence to a conventional happens-before race detector.*

*Our design has modest hardware complexity: caches are completely unmodified and we piggy-back on existing coherence messages but do not otherwise modify the protocol. Furthermore,*RADISH *can leverage type-safe languages to reduce overheads substantially. Our evaluation of a simulated 8-core* RADISH *processor using PARSEC benchmarks shows runtime overheads from negligible to 2x, outperforming the leading software-only race detector by 2x-37x.*

1 Introduction

Data-race freedom is an important safety property for multithreaded programs. Many multithreaded programming errors stem from data races. Memory consistency models of mainstream languages guarantee sequential consistency for data-race-free programs, leaving racy programs with weaker or undefined semantics. Moreover, there is no easy way to tell when a program is race-free: a racy program runs without any notification when sequential consistency is violated. Many static (*e.g.*, [1]) and dynamic (*e.g.*, [2]) data-race detection algorithms serve as useful debugging aids, but their utility is limited by precision or performance: they are either unsound (missing real races), incomplete (reporting false races), or have high performance overheads.[1]

To simplify the semantics of multithreaded programs with races, researchers have proposed eliminating data races entirely [3], through language restrictions that make writing racy programs impossible [4, 5] and by converting all data races [6] – or only those races that may violate sequential consistency [7, 8] – into fail-stop runtime errors. Our focus is on runtime detection of *all* data races, in order to support arbitrary multithreaded programs and to serve as a foundation for enforcing richer safety properties.

This paper introduces RADISH, a hybrid hardware-software data-race detector that is sound (missing no races) and complete (reporting no false races) with performance suitable for most deployment environments. RADISH is the first race detector to achieve this combination of precision and performance. RADISH uses the same vector-clock approach to precise happens-before race detection as software race detectors like FastTrack [2], and provides high performance by storing a useful subset of race detection metadata on-chip in a hardware-managed format. This on-chip metadata allows most race checks to occur completely in hardware with low latency. A simple software layer is responsible for persisting metadata when it overflows hardware's resources, and for using this metadata to check for races when hardware metadata is insufficient.

To help reduce the number of data-race checks that it must perform, RADISH uses cache coherence to detect when threads share data. To maintain precise data-race detection at the byte level, RADISH augments coherence messages with per-byte access history information, but requires no changes to the actual coherence protocol. To keep hardware complexity modest, the only additions to each core are a small amount of state and logic for fast SIMD-style vector-clock computations. Crucially, and un-

[1] We use the definitions of "sound" and "complete" common in the programming languages community.

like many previous hardware proposals, the design of the timing-sensitive cache hierarchy is *entirely unchanged* by RADISH and there is *no dedicated hardware storage* for per-address race-detection metadata. While on-chip and managed by hardware, per-address metadata is stored in dynamically allocated cache lines that share the cache data array with regular data. Thus there is no wasted hardware storage capacity when running programs that do not require dynamic race detection (*e.g.*, due to race-free programming models). Furthermore, RADISH can leverage type safe languages to further reduce overheads.

Our evaluation shows that RADISH provides sound and complete dynamic data-race detection with runtime overheads from negligible to 2x. We also show that, due to the presence of metadata in on-chip caches, increasing cache capacity is a straightforward and effective way of improving RADISH's performance.

The remainder of this paper is organized as follows: Section 2 discusses why precise and inexpensive data-race detection is a useful primitive for the analysis and enforcement of many higher-level multithreaded safety properties. Section 3 reviews happens-before race detection with vector clocks and analyzes the overheads in software-based race detection. Section 4 describes the RADISH algorithm along with its hardware and software components, and Section 5 describes some useful optimizations. Section 6 evaluates RADISH's performance via simulation. Finally, we discuss related work (Section 7) and conclude (Section 8).

2 Why Data-Race Detection?

Data races are typically the result of a programming error, and finding races is a good way to find bugs in programs. Besides this obvious use for precise always-on data-race detection, there are two additional benefits. First, prohibiting data races[2] enables simpler and stronger memory models than current mainstream languages provide. Second, data-race detection is a foundation for enforcing and verifying higher-level concurrency safety properties like atomicity and determinism.

2.1 Simpler Memory Models

Efforts to formalize the memory models of Java [9] and C++ [10] have shown that providing reasonable semantics in the presence of data races is particularly complicated. C++ "avoids" this problem by leaving racy programs with undefined behavior. Attempts to provide guarantees for racy Java programs have uncovered bugs in modern JVM implementations and ambiguities in the Java Memory Model itself [11]. In contrast, an always-on race detector like RADISH guarantees atomicity and isolation of large "interference-free regions" of code across synchronization points [12], a stronger property than sequential consistency.

[2] Annotations can be used to disable race detection for intentional data races, *e.g.*, in lock-free data structures.

2.2 Enforcing Richer Safety Properties

Systems that verify or enforce safety properties like atomicity and determinism typically interpose on program execution only on synchronization operations and data races. However, data-race detection must be fully precise because missing a race can violate the safety property in question, and reporting false races can result in spurious verification failures. Therefore, these systems [13, 14] typically implement a full race detector, resulting in overheads too high for use in deployment. A high-performance data-race filter that avoided instrumentation for data-race-free accesses would make always-on operation practical.

Sound and complete **atomicity specification checking** [13], for example, only needs to instrument synchronization operations and data races. A handler that fires on data races can be used to determine when serializability is violated, if races are not regarded as fail-stop errors. **Transactional memory** [15, 16] would require buffering support via some orthogonal mechanism, but precise conflict detection could be handled entirely by data-race detection. Races would trigger a rollback instead of a program error. **Determinism enforcement** [17] would require only linking with a new synchronization library; races indicate sources of nondeterminism that can be regarded as errors, resolved deterministically, or logged for subsequent deterministic replay. **Determinism checking** [14, 18] would, like determinism enforcement, require action only on synchronization operations. The state space that must be explored while **model checking concurrent programs** is dramatically reduced when races can be ignored [19, 20]. We also envision **new programming models** that leverage language-level data race exceptions (as in [6]) for safer optimistic concurrency, fault recovery, or improved security. Thus, investing hardware resources in efficient, sound, and complete data-race detection would provide a flexible and reusable foundation for a variety of approaches to improving the quality of concurrent software.

3 Happens-Before Race Detection

This section reviews the happens-before race detection algorithm employed (and optimized) by RADISH and prior sound-and-complete data-race detectors. We then discuss experiments we performed profiling the runtime overheads of a state-of-the-art software race detector to highlight the potential performance improvements of hardware-accelerated race detection.

3.1 Happens-Before Race Detection

The *happens-before* relation $\xrightarrow{hb}$ is a partial order over events in a program trace [21]. Given events a and b, we say a *happens before* b (and b *happens after* a), written $a \xrightarrow{hb} b$, if: (1) a precedes b in program order in the same thread; or (2) a precedes b in synchronization order, *e.g.*, a lock release

$rel_t(m)$ and subsequent acquire $acq_u(m)$; or (3) (a, b) is in the transitive closure of program order and synchronization order. Two events not ordered by happens-before are *concurrent*. Two memory accesses to the same address form a *data race* if they are concurrent and at least one is a write.

Vector clocks can track the happens-before relation during execution [22, 23]. A vector clock v stores one integer logical clock per thread. There are two key binary operations on vector clocks: *union* is the element-wise maximum of two vector clocks ($v_1 \sqcup v_2 = v_3$ s.t. $\forall t \,.\, v_3(t) = \max(v_1(t), v_2(t))$); *happens-before* is the element-wise comparison of two vector clocks ($v_a \sqsubseteq v_b$ is defined to mean $\forall t \,.\, v_a(t) \leq v_b(t)$). A vector-clock race detector keeps four kinds of state: Vector clock C_t stores the last time in each thread that happens before thread t's current logical time. Vector clock L_m stores the last time in each thread that happens before the last release of lock m. Vector clock W_x stores the time of each thread's last write to address x. Vector clock R_x stores the time of each thread's last read of address x *since the last write by any thread*. If thread t has not read x since this write, then $R_x(t) = 0$.

Initially, all L, R, and W vector clocks are set to v_0, where $\forall t \,.\, v_0(t) = 0$. Each thread t's initial vector clock is C_t, where $C_t(t) = 1$ and $\forall u \neq t \,.\, C_t(u) = 0$. On a lock acquire $acq_t(m)$, we update C_t to $C_t \sqcup L_m$. By acquiring lock m, thread t has synchronized with all events that happen before the last release of m, so all these events happen before all subsequent events in t. On a lock release $rel_t(m)$, we update L_m to C_t, capturing all events that happen before this release. We then increment t's entry in its own vector clock C_t to ensure that subsequent events in t do not appear to happen before the release t just performed. On a read $rd_t(x)$, we first check if $W_x \sqsubseteq C_t$. If this check fails, there is a previous write to x that did not happen before this read, so there is a data race. Otherwise, we set t's entry in R_x to t's current logical clock, $C_t(t)$. On a write $wr_t(x)$, we check if $W_x \sqsubseteq C_t$ and $R_x \sqsubseteq C_t$. If this check fails, there is a previous access to x that did not happen before this write, so there is a data race. Otherwise, we clear all last reads, setting R_x to v_0, and replace the last write with the current write, such that $W_x(t) = C_t(t)$ and $\forall u \neq t \,.\, W_x(u) = 0$.

By the definition of a data race, all writes to an address must be totally ordered in race-free traces. Race detectors such as Goldilocks [6] and FastTrack [2] leverage this observation by storing information about only a single last write per address. For simplicity of presentation, we retain a last-write vector clock for each address, but observe that it never has more than one non-zero entry.

3.2 Software Race Detection Overheads

Software-only data-race detectors [2, 6, 24, 25] are often too slow for always-on use (Figure 1). The overhead can be attributed to four broad categories: (1) the raw *computation* cost of running the race detection algorithm; (2) enforcing *atomicity* of each memory access and its race check (a lack of atomicity leads to both false and missed races); (3) the *instrumentation* costs of injecting race checks for every memory access; and (4) *other* costs such as the cache pollution of race-detection metadata. We approximated these costs for the state-of-the-art FastTrack dynamic race detector [2], by running the Java Grande benchmark suite on an 8-core machine with different parts of the race detection code disabled (Figure 1). We found that the costs of computation, atomicity, and instrumentation are dominant, though in varying degrees across benchmarks. RADISH directly addresses these costs: the atomicity of race checks is enforced by a lightweight hardware mechanism, race-check computations are frequently done without software assistance, and instrumentation cost is low due to hardware support.

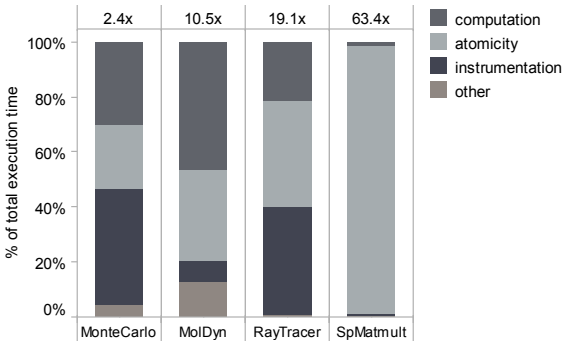

Figure 1. FastTrack execution profiles. Numbers atop each bar indicate FastTrack's slowdown over native execution.

4 The RADISH System

This section discusses the intuition behind RADISH. Then, we describe what RADISH adds to a conventional processor design (Section 4.2), the metadata that RADISH uses (Sections 4.3 and 4.4), the operations RADISH performs at each memory access (Section 4.5), and the RADISH software layer (Section 4.6). We conclude with a short example of RADISH's operation (Section 4.7).

4.1 The Intuition Behind RADISH

The intuition behind RADISH's hybrid hardware-software approach to data-race detection is to start with a software vector-clock data-race detector and map its most heavily used operations and metadata to hardware as frequently as possible. RADISH leverages three basic observations to accomplish this: (1) nearly all the work that a race detector must do occurs in response to coherence traffic so the RADISH mechanisms are rarely activated outside these high-latency events; (2) the spatial and temporal locality exhibited by memory references extends to the metadata necessary for race detection, so the existing cache hierarchy can be used to accelerate metadata accesses; and (3) there is temporal locality in the data referenced by concurrently scheduled threads, so while RADISH only caches metadata

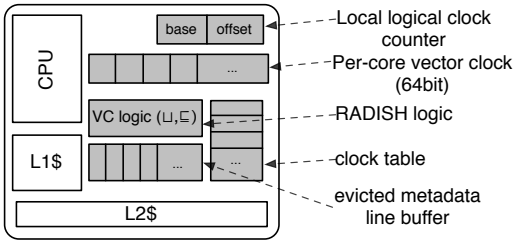

Figure 2. Overview of the RADISH processor core. State added by RADISH is shaded.

for co-scheduled threads this is often sufficient to handle race checks completely in hardware.

We start from the vector-clock algorithm described in Section 3.1 and map parts of it into hardware structures as follows. Each core keeps a portion of its local vector clock C_t on chip, with an entry for each other actively executing thread. Thus, the size of this partial vector clock is bounded by the number of processors. When thread t accesses byte x and needs to access the metadata for x, t's entries $R_x(t)$ and $W_x(t)$ in the last-reads and last-write vector clocks for x are stored as metadata in the cache data array itself, using space otherwise available for data, but obviating the need for dedicated storage. Moreover, locations that can be proven race-free (*e.g.*, local variables) require no metadata, freeing cache capacity for other data or metadata. Lock vector clocks (L_m) are handled purely in software since they are accessed infrequently.

Since RADISH stores only a subset of the full metadata needed for race detection on-chip (only the vector clocks for actively executing threads, and only a subset of read and write vector clocks for these threads, are stored on-chip), a crucial question is how to reason soundly about races from this partial view. We solve this problem with three insights. First, we maintain *in-hardware status* information for each location x that is cached on-chip, summarizing the number and type (read/write) of x's vector-clock entries that are cached in hardware. Second, we rely on software to virtualize limited hardware resources, by storing and providing access to metadata upon last-level cache evictions and context switches. Finally, we memoize the result of vector-clock computations using *local permissions*. This is particularly helpful for race checks performed in software, because they are expensive and require metadata not resident in hardware caches. Memoizing their results as permissions helps avoid repeated expensive checks. The rest of this section explains in detail how RADISH implements these solutions.

4.2 The RADISH Architecture

RADISH makes only minimal changes to a conventional bus-based CMP architecture. Figure 2 shows the additional state added by RADISH with shaded blocks. A per-core vector clock contains a 64-bit clock for each processor in the system, including the local processor. The clock table manages the vector-clock values used by hardware; for efficiency, RADISH employs a reference-counting scheme that we describe later (Section 5.1). Finally, the RADISH logic implements the RADISH algorithm, including the vector-clock operations union ($\sqcup$) and happens-before ($\sqsubseteq$), which can take advantage of SIMD parallelism. RADISH provides atomicity for each memory access, including its corresponding metadata access and race check, by detecting concurrent remote data accesses to the same location. Any such remote access must be the result of a data race. This mechanism uses RADISH's existing precise byte-level communication tracking.

Crucially, RADISH does not change the structure or timing of any portion of the cache hierarchy. Metadata is stored in the caches just like regular data is, and competes for cache capacity just like regular data does. This design choice ensures the critical path latency of cache hits is unchanged, and also ensures that the processor runs with full cache capacity when RADISH is disabled if race detection is not wanted for an application. As metadata is allocated dynamically, any static information about race-freedom can reduce RADISH's space and runtime overheads even further.

4.3 RADISH Metadata

RADISH maintains metadata for each location of virtual memory (discussed below), as well as the per-core partial vector clock mentioned previously. This per-core vector clock is accessible to software, as synchronization operations must read and write it. The vector clocks associated with synchronization objects are accessed relatively infrequently and thus can be managed by a RADISH-aware synchronization library.

RADISH uses hardware to cache a subset of the full metadata needed for race detection, and uses software to persist vector clock entries when they are evicted from the cache hierarchy. There can thus be two versions of any given entry of a vector clock – one in hardware and another in software. There are no version conflicts because hardware is always most up-to-date. Appendix A.1 shows that RADISH's two vector clock versions can always be reconciled to the values a conventional vector clock race detector maintains.

RADISH maintains metadata for each byte in memory, since that is the finest granularity at which a program may access memory, according to modern memory models [10, 9], though RADISH can exploit the type safety guarantees of modern languages to soundly coarsen the metadata granularity for improved performance (Section 5.2). Without reliable information on data element size, tracking accesses with metadata for, *e.g.*, every 2 bytes could find false races if two threads concurrently write to the two different bytes.

The RADISH metadata for a byte of data consumes a total of 2 bytes of space (Figure 3), though a more compact 1:1

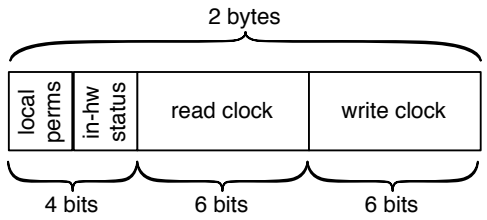

Figure 3. RADISH**'s in-cache metadata format. Local permissions have 3 possible values, and in-hardware status 4 values, so we use 4 bits to represent the 12 combinations. 6 bits are left for each clock; we discuss rollover issues in Section 5.1.**

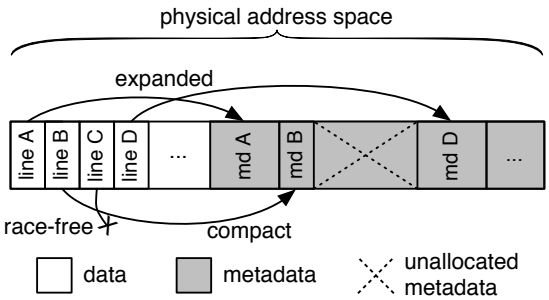

Figure 4. Mapping from data to metadata addresses for a single processor. Each processor uses a distinct portion of the physical address space.

encoding is possible by leveraging type safety guarantees (Section 5.2). 2 bytes of metadata per byte of data admits a simple mapping from data to metadata: the location of the metadata for address x is located at address $base + 2x$, where $base$ is chosen not to overlap with regular program data (Figure 4). Metadata addresses are special physical addresses that only ever reside in the cache tag arrays – hardware's metadata does not occupy any physical memory, as that would be redundant with software's representation. The metadata for a cache line's worth of data is split across two cache lines (as with cache lines A and D). These two metadata lines need not reside in the cache at the same time – metadata is fetched on demand based on the corresponding data being accessed. Other lines may use the compact metadata encoding (B) or may not need metadata at all (C) courtesy of a previous static analysis. The "holes" in the data-to-metadata mapping are unallocated and never fetched into the cache, making room for other useful (meta)data.

Each processor uses a distinct region of the physical address space for metadata. We can efficiently encode the fact that a line contains metadata, and also the processor p to which the metadata belongs, by stealing some high-order bits from the physical address. This ownership information allows metadata to live in shared caches. Upon eviction from the last-level cache, metadata is stored by software, which uses its own opaque format that occupies virtual memory just like regular program data. Software-controlled metadata is never touched by hardware, so its format can be tuned to a particular run-time system, programming model, or even application, for maximum efficiency.

In addition to read and write clocks, the RADISH metadata consists of two additional pieces of state (Figure 3): in-hardware status and local permissions. This additional metadata is crucial for getting the most leverage from the metadata cached in hardware, so as to avoid consulting software in common cases.

In-hardware status encodes how much of the metadata for a given location resides in hardware. The status can be one of: LASTWRITE, which says that the metadata recording the most recent write to a location is in cache; ALLLASTREADS, which says that the metadata recording all of the most recent reads of a location are in cache; EVERYTHING, which says that information about both the last reads and last write is in cache; or finally INSOFTWARE, which says that software must be consulted to determine the most recent accesses for this location. RADISH uses in-hardware status information to determine when a cache fill or race check, which would otherwise require consulting software, can in fact be done entirely in hardware.

Local permissions specify the actions that the local thread can perform on a location without being involved in a data race. The allowed actions are encoded as *permissions*: a thread has READ, WRITE or NONE permissions to a location. WRITE implies READ. Local permissions act as a filter, guaranteeing that no race check is necessary if local permissions are not violated. This allows RADISH to avoid performing a race check (whether in software or fully in hardware) on every memory access. Permissions violations may or may not be the result of an actual race; additional work is required to disambiguate these cases.

Metadata lines are not subject to the normal coherence protocol, but are instead updated on evictions of remote metadata lines, when local permissions are violated, and whenever coherence events take place on their associated data line. This latter property allows many metadata updates to piggy-back on regular coherence messages; metadata evictions and local permissions violations are the only sources of extra interprocessor communication in RADISH.

4.4 Maintaining In-Hardware Status and Local Permissions

On a write, the **in-hardware status** for the bytes being written is set to EVERYTHING. The EVERYTHING status propagates via coherence messages to subsequent reads and writes of x, and is only demoted when metadata for one of these reads or writes to x is evicted, as detailed in Figure 5. Note there cannot be a valid write clock evicted while the status is ALLLASTREADS: since the valid last write a was already evicted, another write a' would need to occur, but a' would have reset the status back to EVERYTHING. The in-hardware status can also be set on reads, if the read

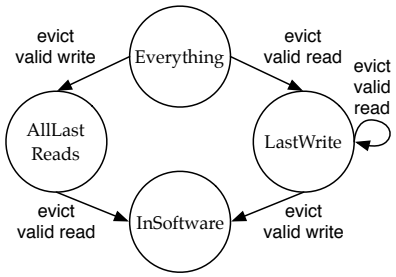

Figure 5. In-hardware status is downgraded when metadata is evicted from the last-level cache.

triggers a software vector-clock check. If the check reveals that thread t is the sole last reader of a location x since the last write, then t's read means the in-hardware status can now be set to ALLLASTREADS. In RADISH, metadata evictions from the last-level cache require a broadcast to downgrade in-hardware status appropriately, but this cost can be masked by the L2 miss that triggered the LLC eviction.

Local permissions are set for each byte x in a line when it is brought into t's cache, by broadcasting (concurrently with the fill) to gather the last accesses to x by other scheduled threads. t then checks its happens-before relation with these remote accesses. Thread t gets WRITE permission for x if t's current logical time happens after the last write to x and all last reads of x (*i.e.*, $W_x \sqsubseteq C_t$ and $R_x \sqsubseteq C_t$), READ if t's current logical time happens after the last write to x but not all last reads of x (*i.e.*, $W_x \sqsubseteq C_t$ and $R_x \not\sqsubseteq C_t$), and NONE otherwise. In-hardware status determines whether the on-chip metadata suffices to attempt each of these checks. If byte x has insufficient in-hardware metadata, RADISH sets NONE permission for x; this is conservative but cheaper than consulting software to get a precise result, as t may never access x.

When a remote thread u performs a read or write of x, we need to update t's local permission for x to maintain its guarantee. If u does a write to x, we must "downgrade" t's permission on x to NONE. If u does a read, then t's permission must be downgraded to READ as well, since t's current logical time no longer happens after all last accesses to x, making a write unsafe. If t's permission for x was previously NONE, it is unchanged – downgrading never increases permissions. It is sound to perform downgrades only on coherence events and cache fills as shown in Appendix A.2.

4.5 RADISH Runtime Checks

Race checks in RADISH are a 3-stage process: a permissions check, a hardware race check, and a software race check (Figure 6). First, a thread t accesses a memory location x. The load or store instruction may be marked statically as race-free (*e.g.*, a compiler may tag accesses to non-escaping local variables), or the location accessed may reside on a page tagged as containing only race-free locations (*e.g.* thread-local storage); in these cases no further work is necessary. Otherwise, we consult t's metadata for x; if the metadata is not in the local processor's cache it is fetched from other caches or software (Section 4.6).

Once t's metadata for location x is in cache, the first step of a RADISH race check is a **permissions check**: if local permissions allow the access, no further race checking is necessary. A **hardware race check** is performed if the permissions do not allow the access to proceed. A hardware race check consults the precise read/write clock values from other caches, together with the local in-hardware status metadata, to determine if the access to x is race-free. Specifically, a read operation r is race-free if (1) r happens after some last-read operation s (since s must happen after the last write to x or a race would have been detected on one of those previous accesses) or (2) the in-hardware status for x is LASTWRITE and r happens after the last write. A write operation w is race-free if and only if the in-hardware status for x is EVERYTHING, and w happens after the last write and all last reads.

Depending on the amount of state available in hardware, the outcome of the hardware race check may be that a) there definitely is a race, b) there definitely is not a race, or c) there might be a race, *e.g.*, there is no race with respect to the in-cache data, but the in-hardware status is INSOFTWARE so there is additional relevant information in software. A **software race check** is required only in case c). To exploit spatial locality and amortize the overhead of invoking software, the software check performs a race check for all data locations covered by the metadata line ℓ containing the metadata for x. These checks preemptively set local permissions for all metadata in ℓ.

4.6 The RADISH Software Interface

For RADISH, the system's **synchronization library** must be modified to update the per-core vector clocks on synchronization operations, and to maintain a vector clock with each synchronization object. There are also software handlers for **race checks** and **metadata evictions**. The race check handler is called on a memory operation when hardware does not have enough information to prove race-freedom. The race check handler may be called synchronously or asynchronously. The eviction handler is called when metadata is displaced and may be executed asynchronously with respect to memory operations. Thus, these handlers may execute on any available processor.

A **software race check handler** is passed, via registers, the physical address p_x of the location x that triggered the check, as well as the current hardware entries for the read/write vector clocks of x. p_x is used to index the *variable map*: the central software data structure used by the RADISH software layer. The variable map contains a mapping from physical addresses (of data) to software metadata (*e.g.*, read/write vector-clock pairs). Physical addresses are used because they are convenient identifiers – since caches

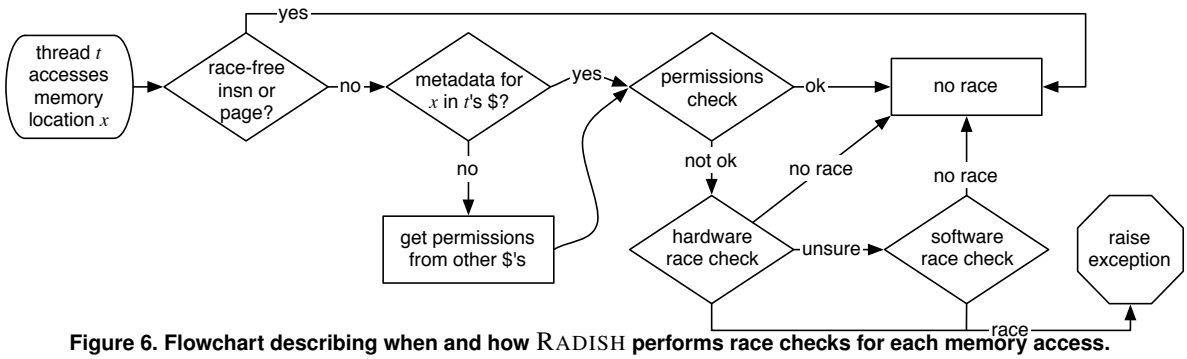

Figure 6. Flowchart describing when and how RADISH performs race checks for each memory access.

Event	Software		p0 running thread 0					p1 running thread 1					In-hardware
	read vc	write vc	coh. state	perms	read	write	vc	coh. state	perms	read	write	vc	Status
(initial)	7,0,0	2,0,0					$\underline{9}$,4					5,$\underline{6}$	InSoftware
p0 read			S	W	9	2							Everything
p1 read				W				S	R	5	0		
p0 release							$\underline{10}$,4						
p1 acquire												5,$\underline{9}$	
p1 write			I	N	0	0		M	W	0	5		
p0 evict md			-										
p1 evict md	0,0,0	0,5,0						-					InSoftware

Table 1. An example trace showing how RADISH metadata is updated. Empty cells indicate the value is the same as in the cell above. The local component of each core's vector clock is underlined.

are physically tagged, when evicting a metadata cache line we identify it with its physical address to avoid any need for reverse translation. These physical addresses do not allow the program unmediated access to physical memory. When the OS migrates physical pages, the variable map must be updated accordingly, but a physically-indexed map keeps the common case fast.

A software race check merges the hardware read/write vector clock entries with the values from software (obtained from the variable map) to obtain up-to-date read/write vector clocks (R_x and W_x in Section 3.1). Similarly, the entries from the per-core partial vector clock (accessible via new instructions) are used to obtain the up-to-date thread's vector clock (C_t). Once hardware and software values are merged, the standard happens-before race check occurs. A signal may be raised to indicate that a race has occurred.

To amortize the cost of invoking software, software race checks verify the race-freedom of a single memory access, but update hardware metadata for all locations in the cache line with metadata from software and set the in-hardware status for each location where software information is imported into hardware.

The **eviction handler** is called whenever a metadata line is evicted from the last-level cache. This handler is invoked with the physical address p of the data corresponding to the metadata being evicted; p is used to index the variable map described previously. To process metadata evictions asynchronously, evicted metadata lines are placed into a hardware buffer. The eviction handler reads from this buffer via special load instructions and updates the variable map with the hardware values.

In RADISH, the software metadata representation is opaque to hardware, allowing software to freely optimize its metadata representation, *e.g.*, to save space [26, 2] or to leverage structured parallelism [27]. A software-managed metadata format also admits compatibility with compacting garbage collectors that move objects in physical memory.

On a **context switch**, per-core partial vector clocks must be flushed into software, and all per-core clocks need to be updated to replace entries for the descheduled thread with entries for the incoming thread. The evicted metadata line buffer can be cleared lazily, as entries are tagged with their owner thread. The crucial issue is dealing with metadata lines belonging to the descheduled thread, as they can occupy several MB of state. We observe that software can conservatively approximate when its metadata is stale (*i.e.*, when there exists metadata in hardware that is more up-to-date) by setting a "potentially stale" bit whenever a software check occurs (as a check must occur on the very first access to a location, which brings metadata into hardware), and clearing this bit when sufficient metadata has been evicted.

Using software's conservative approximation of hardware state, metadata lines can be flushed eagerly or lazily. The eager approach uses software to flush these lines immediately at the context switch, bringing all software metadata up-to-date. Alternatively, flushing can be done lazily. At the cost of stealing extra unused bits from the physical address space, each thread ID can be allocated a region within the metadata space of a processor. Thread IDs can be reused across processes: as metadata lives in the physical address space, the same thread ID in two different processes will be isolated if the processes are isolated.[3] When asked to per-

[3] If processes share memory, then shared regions must be flushed when-

form a race check, software determines if its metadata is potentially stale, and flushes a superset of the necessary entries (even for descheduled threads) from hardware caches. With lazy flushing, clock tables are saved on context switches to ensure software interprets hardware metadata correctly.

4.7 An Example Trace

We now show RADISH's operation on a short trace of instructions. Table 1 shows how hardware metadata (local permissions, read/write vector clock entries, per-core vector clocks, and in-hardware status) and software metadata (only read/write vector clocks are shown for simplicity) are updated in response to various events. For simplicity, events for a single location x are shown. There are three threads, but only threads 0 and 1 are scheduled (on processors p0 and p1, respectively).

Initially, there is no metadata cached in hardware. p0's load triggers a software race check, because the in-hardware status INSOFTWARE indicates that there is insufficient information in hardware to check the access. The software check brings information about p0's previous write at time 2 into hardware; combined with p0's read there is now complete information about x in hardware and the in-hardware status is upgraded to EVERYTHING. Furthermore, p0 obtains WRITE permissions on x because p0 can soundly read or write x without the need for further race checks. Next, p1 performs a read that, due to the current in-hardware status, can be checked without consulting software. Note that p1 gets READ permissions, as it can perform further reads of x without racing, but p0 retains its WRITE permissions (though an immediate write by p0 would race). However, since p0 has the cache line containing x in Shared state, it cannot actually write to x without triggering a coherence message. This message will downgrade p0's local permissions to READ, so the permissions check will fail and trigger a deeper check that will catch the race.

Next, synchronization occurs from p0 to p1, which updates both processors' vector clocks. Then p1 performs a write, which is well-synchronized with both the last write (by p0) and the last read (also by p0). p1's write downgrades the local permissions for p0, and clears all other hardware read/write clocks. Since p0's clocks have been reset, the read/write vector clocks in software do not change when p0 evicts the metadata for x. Software's metadata is stale, but is brought up-to-date with p1's eviction.

5 Further Optimizations

RADISH's performance is greatly improved by three additional optimizations. We show how to (1) reduce logical clock rollovers, (2) leverage type-safe languages and (3) use spare cores to offload the work of software race checks.

ever a new process is scheduled. The physical address space could be shrunk further to accommodate process IDs, but the overhead (combined with thread IDs) is likely to be too great.

5.1 Logical Clock Rollovers

RADISH metadata affords only 6 bits for storing logical clocks representing a processor's last write or read of a byte. The literal value stored in these 6 bits is interpreted as an offset from a base clock maintained by the processor, with one value reserved to mean "not accessed," which is distinct from "last accessed at $base + 0$." This mechanism can represent last accesses at no more than 63 *consecutive* logical clocks within the cache at any time. To represent a clock outside the available range, we must evict *all* metadata from the cache and reset to a compatible base clock.

To mitigate the performance impact of 6-bit clocks we can interpret the bits as a *pointer* to a clock value instead. We reference-count these pointed-to clocks (using the "clock table" in Figure 2), updating counts when metadata is overwritten or evicted, to determine when they can be recycled. New clocks are allocated on synchronization that increments the local processor's logical clock. The processor now maintains a 63-entry table mapping 6-bit pointers to full logical clocks plus reference counts: using 64-bit logical clocks and 24-bit reference counts, this table occupies 700B. This garbage-collection scheme supports any 63 unique clocks in the cache at a time; they need not be consecutive, so running out is rare (as Section 6 shows).

5.2 Leveraging Type Safety

Type-safe languages provide the guarantee that, between allocation and deallocation (or garbage collection), all accesses to an address x will be at the same granularity; component bytes of multi-byte items are never accessed individually and therefore their metadata is always identical.[4] If RADISH knew this type safety guarantee, it could (1) exploit the presence of multi-byte data items to encode larger read and write clock values to avoid rollover and (2) adopt more compact metadata representations.

Type safety guarantees allow **metadata coarsening**: merging the metadata storage for multi-byte items and storing larger read and write clocks to ease pressure on the available clock range (Section 5.1). To interpret the contents of variable-granularity metadata storage correctly, such as when looking up local read and write clocks in response to another processor's request, granularity information must be persisted with the metadata. In type-safe mode we learn granularity lazily. Before an address has been accessed for the first time, the granularity of metadata does not matter. On the first access to an address, we use the width given by the instruction to set its metadata granularity. The granularity is encoded using the spare states available from encoding in-hardware status and local permissions in 4 bits (Section 4.3), and the additional storage available with coarser-

[4] Casting to differently-sized integers can be handled by performing truncation/expansion only in registers, so that loads and stores always access whole data items.

grained metadata. With this type safety optimization, byte-granularity metadata is reference-counted (Section 5.1), and all coarser-grained metadata is interpreted as an offset from some base logical clock value – reference-counting for large clock representations requires infeasible amounts of hardware state. The extra room for encoding clock values makes rollover an even rarer event.

Type safety's consistent-granularity-access guarantee also enables a more **compact metadata** encoding. If a data line has no 1-byte items, then its metadata can fit in a single cache line instead of two. 16-bit data items have standard 6-bit reference-counted clock values, and larger data items can take advantage of metadata coarsening as before. Since metadata lines do not participate in the coherence protocol, we encode whether a metadata line is in compact or expanded mode with the bits used to maintain the coherence state. Using compact metadata lines does *not* alter RADISH's simple mapping from data to metadata lines; rather it introduces "holes" wherever compact metadata is used (Figure 4). As metadata cache lines are dynamically allocated in RADISH, these holes free up cache capacity.

We can optimistically adopt the compact encoding when a metadata line ℓ is filled without sacrificing correctness. If there are no single-byte data items in ℓ, then the compact encoding is sufficient. Upon the first byte-sized access to ℓ, however, we must revert to the expanded metadata encoding to be able to track byte accesses precisely. At this point, the first such byte access has not yet occurred, so our compact metadata encoding is still precise. RADISH hardware allocates room in the cache for the second metadata cache line ℓ', and then expands the metadata from ℓ to fill both metadata lines. Finally, the byte access proceeds and the metadata is updated precisely.

5.3 Asynchronous Software Checks

Software checks are one of the key sources of overhead in RADISH. Performing them synchronously, *i.e.*, immediately before a potentially racing load or store, provides guaranteed fail-stop semantics but also adds latency to a thread's critical path. Alternatively, these software checks can be performed asynchronously; a thread that requires a software check for location x enqueues a snapshot of x's read and write vector clocks (from hardware), and then proceeds as if the software check had found no race. Spare cores then subsequently perform these checks and validate the race-free assumption. RADISH requires that all outstanding software checks be completed at each synchronization operation and system call,[5] to sandbox the ill effects of races and to avoid the need to snapshot per-core vector clocks or software state. The relaxed guarantees of asynchronous checks support without modification all the checking and enforcement mechanisms we identify in Section 2.2, though future race-recovery-based programming models may require the stricter guarantees of synchronous checks.

[5] In theory, a thread may diverge due to a race and never reach the next synchronization operation, leaving the race unreported. Bounding the latency of asynchronous checks ensures all races are eventually reported.

6 Evaluation

We evaluate RADISH's performance with a Pin-based simulator [28]. We model a multiprocessor with simple cores and a realistic memory hierarchy; memory instructions have variable latency (explained below) and all other instructions take 1 cycle. We model an 8-core system with a MESI coherence protocol, 8-way 64KB private L1's, 8-way 256KB private L2's and a 16-way 16MB shared L3 (all with 64B lines). L1, local L2, remote L2, L3 and memory accesses are 1, 10, 15, 35 and 120 cycles, respectively. Hardware race checks are 10 cycles, and for software race checks we simulate the actual instructions performed by the FastTrack [2] algorithm, which results in variable latency of at least 100 cycles but potentially much higher depending on the amount of data accessed and where it is in the memory hierarchy. Epoch rollovers cost 100,000 cycles. Stack accesses (as identified by Pin) are assumed to be thread-local (to approximate an escape analysis), triggering neither race checks nor metadata lookups. We model the effects of metadata lines occupying cache space, and all extra messages and state that RADISH uses. RADISH's software race checks have a fixed cost for transitioning to software, plus a variable cost for the actual check (we simulate the FastTrack algorithm [2]). There is no transition cost when simulating FastTrack alone. Unless noted, all RADISH experiments use the baseline configuration above, compact metadata, clock value reference counting, type safety optimizations, and synchronous software race checks. We evaluate RADISH on the PARSEC 2.1 benchmarks [29], with simsmall inputs and 8 threads. We report results for a subset of the benchmarks, due to lock-free algorithms for which race detection is not meaningful (canneal), very large simulator memory usage (facesim, ferret, raytrace, freqmine and dedup), and our simulator's lack of support for reader-writer locks (bodytrack). We report performance as the mean of 5 runs; error bars give 95% confidence intervals. Our experimental data and simulator source code are available from `http://sampa.cs.washington.edu`.

We first compare RADISH with a simulated version of the FastTrack [2] sound and complete software-only race detector. In all cases, RADISH has several times less overhead than pure software. Then we examine RADISH's two primary, and interrelated, sources of overhead: the increase in cache pressure due to metadata lines in the cache, and the need for software race checks when there is insufficient metadata cached in hardware. We show that two simple techniques – provisioning extra cores to perform software checks asynchronously and increasing cache capacity – are effective optimizations. We show that RADISH's core race

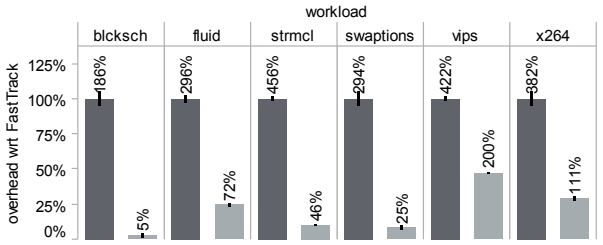

Figure 7. Bars show overhead of FastTrack (dark) and synchronous RADISH (light) normalized to FastTrack. Numbers show overhead with respect to a non-RADISH system.

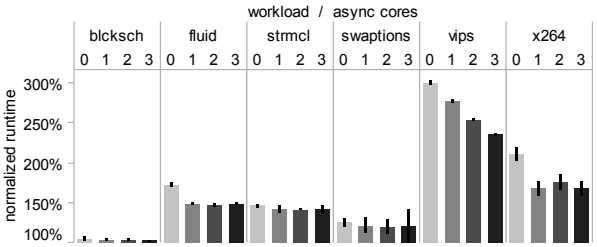

Figure 8. The diminishing returns of increasing the number of cores dedicated to asynchronous checks. The lightest bars ("0") are for synchronous RADISH.

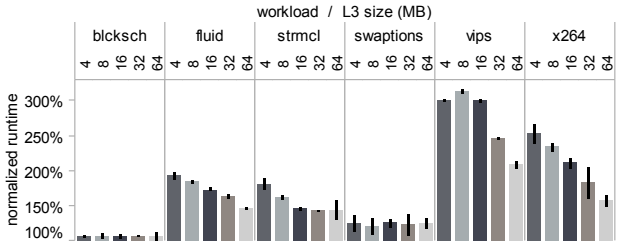

Figure 9. Performance effects of varying last-level cache (L3) capacity, from 4MB to 64MB.

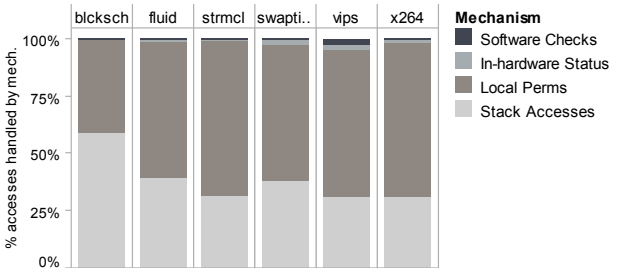

Figure 10. Percentage of memory accesses handled by RADISH's various race detection mechanisms.

detection algorithm consults software rarely, and that type safety optimizations reduce metadata cache footprint and eliminate logical clock rollovers.

6.1 Performance

Figure 7 compares the runtime overhead of FastTrack and synchronous RADISH. Lower is better, and 100% means running as fast as FastTrack. RADISH substantially outperforms the FastTrack detector, reducing the overhead of race detection by 2x (vips) to 37x (blackscholes), while offering the same sound and complete detection guarantees. The numbers above each bar show overhead compared to an equivalent non-RADISH system (*i.e.* no race detection). RADISH's hardware acceleration not only outperforms software but brings race detection overheads to an acceptable level from negligible to 3x. Asynchronous checking (Figure 8) and larger caches (Figure 9) reduce overheads further: dedicating two additional cores to asynchronous checking and leveraging a 32MB L3, RADISH's overheads range from negligible to 2x. RADISH's overheads are low enough to provide the benefits of always-on, precise race detection to many applications.

6.2 Sensitivity Analysis

Figure 8 shows the additional performance obtained by dedicating more cores to processing **asynchronous checks**. Results are normalized to a non-RADISH system with eight cores. The lightest bars show the overhead of having fully synchronous checking. The darker bars show the performance improvement from having 1, 2 or 3 cores dedicated to asynchronously processing the checks generated by the eight application cores. Though most workloads do not benefit from more than one additional core, vips (in experiments not shown) can profitably use up to four extra cores.

Figure 9 shows that the performance of RADISH is highly dependent on **cache capacity**. We focus on the last-level cache since its design is the most amenable to capacity increases. The darkest bars (16M) in Figure 9 show the performance of our baseline RADISH system with a 16MB shared L3. Other bars show the performance of smaller and larger shared L3 caches; line size and associativity, and L1 and L2 private caches, are unchanged. RADISH is able to readily harness extra cache capacity for workloads with larger working sets (fluidanimate, vips and x264). 64MB L3's offer little marginal benefit. In other experiments (not shown), we evaluated RADISH's performance on caches of higher associativity and smaller line sizes (but equivalent capacity) than our baseline and found performance unchanged. Thus, the presence of RADISH metadata in on-chip caches has a noticeable effect only on capacity misses. While RADISH does increase an application's working set, growing the size of on-chip caches (particularly at the last-level) is a common and power-efficient way of spending increasing transistor budgets. Increasing last-level cache capacity is a simple but useful optimization for RADISH.

In Figure 10 we show what percentage of dynamic memory accesses are handled by each of RADISH's race detection mechanisms. Across all benchmarks, a large number of accesses are ignored because we assume that stack accesses are race-free (to approximate the information a compiler could provide about static race-freedom). An even larger

proportion of accesses are proven race-free by the local permissions mechanism. In-hardware status allows accesses to be checked via explicit hardware race checks. While there are many fewer accesses that utilize in-hardware status than the other RADISH mechanisms, software race checks are so expensive that in-hardware status is still a critical performance optimization. The remaining accesses must be checked in software, and this frequency is the key determinant of RADISH's performance.

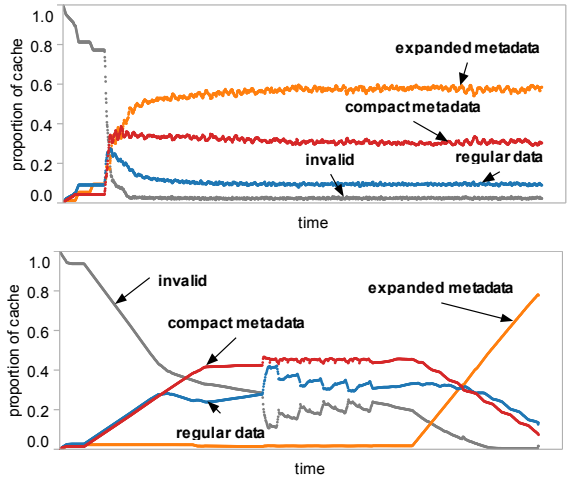

Figure 11. Proportion of hardware caches used for various kinds of data, throughout executions of vips **(top) and** fluidanimate **(bottom).**

To characterize RADISH's space overheads, we measured cache contents (for the entire hierarchy) every 1M cycles and determined how many lines were invalid, or filled with data, compact metadata or expanded metadata. Figure 11(a) shows the results for an execution of vips. After a startup phase, vips utilizes its entire cache capacity, and most is dedicated to metadata. RADISH's compact metadata optimization is effective for many metadata lines, though a large number of byte arrays filled with pixel data (vips is an image processing benchmark) require most metadata lines to be in expanded form. In contrast, in fluidanimate compact metadata is much more common than expanded (the common case for our workloads). The steady rise in expanded metadata at the end of fluidanimate is from writing an output file as a series of character strings. RADISH adapts to fluidanimate's run-time behavior, using compact metadata for most of the execution and dynamically allocating expanded metadata only when necessary.

Finally, Table 2 shows that expensive events are rare in RADISH. Metadata evictions from the last-level cache, metadata inflations triggered by a byte access to a compact metadata line, hardware and software race checks, and epoch rollovers are all rare. The last column quantifies the effect of RADISH's metadata cache pollution on L1 cache misses for regular data accesses. These data show that keeping software race checks infrequent is the key to high performance; the frequency of hardware race checks and degradation of L1 hit rates has much less impact. Also noteworthy is that no benchmarks experienced an epoch rollover during our experiments – our rollover optimizations eliminate this potential source of overhead.

workload	md evicts	md inflates	race checks hw	race checks sw	roll-overs	L1D misses (base/RADISH)
blacksch	0.0	0.0	1.6	0.1	0.0	8.5 / 9.5
fluid	1.2	0.0	2.2	1.4	0.0	10.0 / 15.3
streamcl	0.0	0.0	4.4	0.1	0.0	63.8 / 66.1
swaptions	0.0	0.0	8.6	0.1	0.0	21.6 / 30.1
vips	6.6	0.3	9.5	9.3	0.0	30.3 / 56.3
x264	1.8	0.7	4.7	2.3	0.0	18.4 / 29.7

Table 2. Frequency of important RADISH events, expressed as occurrences per 1,000 instructions.

7 Related Work

We now discuss other related work on hardware support for dynamic race detection; space constraints preclude a discussion of static race detectors or detectors for other concurrency errors. Conflict Exceptions [7] and DRFx [8] are especially related to RADISH. They generate exceptions only for races that may violate sequential consistency in data-race-free models. DRFx uses a hardware buffer of memory locations accessed between fences and coherence event monitoring that checks for conflicts with addresses in the buffer. Conflict Exceptions keeps pre-assigned byte-level access bits per cache line and sets aside memory space to keep access bits for out-of-cache data. Both proposals have large dedicated hardware structures: Conflict Exceptions adds 50% cache overhead, and DRFx adds 10KB of hardware state. The HardBound system [30] also leverages the idea of storing metadata in the cache itself, albeit only for shared caches and also in the context of providing memory safety for C programs. Aikido [31] uses dynamic binary rewriting and hardware memory protection mechanisms to detect sharing more efficiently than pure software, accelerating shared-data dynamic analyses such as race detection.

Min and Choi [32] developed a limited form of happens-before race detection using coherence events for programs with structured parallelism. SigRace [33] uses signatures to accelerate race checks; it employs a checkpoint/rollback mechanism to re-execute when a conflict is detected to prune some false positives. Still, SigRace can report false races due to signature imprecision and granularity of access monitoring, as well as missed races due to limited buffer space for checkpoint/rollback. CORD [34] approximates happens-before race detection using per-word vector clocks (fixed metadata) for in-cache data only, leading to both unsoundness and potential incompleteness. ReEnact [35] uses thread-level speculation mechanisms to detect races and potentially recover from them via checkpoint/rollback. ReEnact can report false races due to word-granularity tracking

and can miss races due to finite hardware resources.

Eraser [36] proposed lock-set violation detection, an approximation of happens-before race detection that suffers from false positives. HARD [37] is a hardware-based implementation of the lock-set algorithm that uses bloom-filters per cache line to encode which locks should be held when accessing the corresponding data. While locking-discipline violation detection is very useful for debugging, it is imprecise for many acceptable programming idioms.

ECMon [38] proposes exposing cache coherence events to software as a primitive for several program monitoring and control techniques. It shows how to use these primitives for deterministic record/replay as well as speculative barriers. ECMon has a subset of the support that RADISH offers, as we need hardware for byte-level metadata tracking and vector-clock comparisons, since trapping to software for every metadata update and every vector-clock computation would be prohibitively expensive.

8 Conclusions

Sound and complete race detection is an important mechanism for detecting bugs, simplifying memory model semantics, and providing parallel programming safety properties like atomicity and determinism. Existing software approaches to race detection have high overheads, and existing hardware approaches either miss races, report false races, or both. We propose RADISH, the first hardware-accelerated race detection algorithm that is sound, complete, and fast enough for always-on use. Through simulation, we show that RADISH has no more than 2x overhead compared to normal execution, and outperforms the leading software-only race detector by 2x-37x.

References

[1] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, 2003.

[2] C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[3] S. Adve and H. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, Aug 2010.

[4] G. Blelloch. NESL: A Nested Data-Parallel Language. Technical report, Pittsburgh, PA, USA, 1992.

[5] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[6] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, 2007.

[7] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, 2010.

[8] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, 2010.

[9] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *POPL*, 2005.

[10] H. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.

[11] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*. 2008.

[12] L. Effinger-Dean, H. Boehm, P. Joisha, and D. Chakrabarti. Extended Sequential Reasoning for Data-Race-Free Programs. In *MSPC*, 2011.

[13] C. Flanagan, S. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, 2008.

[14] C. Sadowski, S. Freund, and C. Flanagan. SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In *ESOP*, 2009.

[15] K. Moore, J. Bobba, M. J. Moravam, M. Hill, and D. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, 2006.

[16] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA*, 2005.

[17] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[18] M. Bender, J. Fineman, S. Gilbert, and C. Leiserson. On-the-fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *SPAA*, 2004.

[19] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[20] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, 2010.

[21] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21, July 1978.

[22] C. Fidge. Logical Time in Distributed Computing Systems. *Computer*, 24, August 1991.

[23] F. Mattern. Virtual Time and Global States of Distributed Systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[24] E. Pozniansky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *PPoPP*, 2003.

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

[26] M. Christiaens and K. de Bosschere. Accordion Clocks: Logical Clocks for Data Race Detection. In *Euro-Par*, 2001.

[27] J. Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *SC*, 1991.

[28] C. K. Luk et al. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[29] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[30] J. Devietti, C. Blundell, M. Martin, and S. Zdancewic. HardBound: Architectural Support for Spatial Safety of the C Programming Language. In *ASPLOS*, 2008.

[31] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe. Aikido: Accelerating Shared Data Dynamic Analyses. In *ASPLOS*, 2012.

[32] S. Min and J.-D. Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *ASPLOS*, 1991.

[33] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, 2009.

[34] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *HPCA*, 2006.

[35] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *ISCA*, 2003.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ToCS*, 15(4), 1997.

[37] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, 2007.

[38] V. Nagarajan and R. Gupta. ECMon: Exposing Cache Events for Monitoring. In *ISCA*, 2009.

A Correctness

In this section, we show the soundness and completeness of RADISH via its equivalence to a vector-clock race detector, known to be sound and complete [22, 23]. We first show how RADISH's hybrid hardware and software metadata is equivalent to the metadata stored by a vector-clock race detector and how the algorithm preserves this equivalence. Then we show that the in-hardware status and local permissions optimizations are sound. We model RADISH with no context switches, so threads and processors are equivalent and we refer to them as threads. We discuss context switches, and other systems issues, in Section 4.6.

A.1 Equivalence to a Vector-Clock Race Detector

RADISH stores the full vector-clock race detector state (C, L, R, W) (as introduced in Section 3.1) as a hybrid of hardware metadata $(\mathbb{C}, \mathbb{R}, \mathbb{W})$ and software metadata $(\mathcal{C}, \mathcal{L}, \mathcal{R}, \mathcal{W})$. We use the following notation throughout: $\mathbb{C}_t(u)$ is thread u's entry in thread t's on-chip vector clock. $\mathbb{R}_x(t)$ and $\mathbb{W}_x(t)$ are byte x's last-read and last-write clocks in thread t's cached metadata. $\mathcal{C}_t(u)$ is thread u's entry in thread t's SW vector clock. $\mathcal{R}_x(t)$ and $\mathcal{W}_x(t)$ are thread t's entries in byte x's last-reads and last-write SW vector clocks. $\mathcal{L}_m(t)$ is thread t's entry in lock m's SW vector clock.

RADISH's hybrid state (equivalent to the full vector-clock race detector state) is the software state, with elements overridden by hardware state where available as follows: $C_t =$ if $t \in \mathsf{Dom}(\mathbb{C})$ then $\mathbb{C}_t$ else $\mathcal{C}_t$. $L_m = \mathcal{L}_m$. $R_x(t) =$ if $t \in \mathsf{Dom}(\mathbb{R}_x)$ then $\mathbb{R}_x(t)$ else $\mathcal{R}_x(t)$. $W_x(t) =$ if $t \in \mathsf{Dom}(\mathbb{W}_x)$ then $\mathbb{W}_x(t)$ else $\mathcal{W}_x(t)$.

In the unoptimized version of RADISH there is no in-hardware status or local permissions, so we must perform a race check on every memory operation. On an access to address x by thread t, metadata for address x is loaded into cache from software if it is not already in cache. The last-read and write clocks $\mathbb{R}_x(t)$ and $\mathbb{W}_x(t)$ are initialized with t's entries $\mathcal{R}_x(t)$ and $\mathcal{W}_x(t)$ in the software last-reads and last-write vector clocks. If t's metadata for x is evicted from cache, $\mathbb{R}_x(t)$ and $\mathbb{W}_x(t)$ are persisted back to software vector clocks in $\mathcal{R}_x(t)$ and $\mathcal{W}_x(t)$, respectively. Whenever the metadata is in hardware, it overrides software metadata, so stale software metadata is never observed.

On every memory access to x by t, we perform the $W_x \sqsubseteq C_t$ check and, for writes, the $R_x \sqsubseteq C_t$ check from the vector-clock race detector. Since the metadata are equivalent, the checks will always have the same result when checking the same operations.

On a race-free write by thread t, we set the metadata clocks $\mathbb{R}_x(u)$ and $\mathbb{W}_x(u)$ to zero for all threads u that have x's metadata in cache. For those threads u that do not have x's metadata in cache, we zero the entries in the software vector clocks $\mathcal{R}_x(u)$ and $\mathcal{W}_x(u)$. This is equivalent to zeroing all entries in full vector clocks R_x and W_x, by the "hardware overrides software" rule. We set $\mathbb{R}_x(t)$ or $\mathbb{W}_x(t)$ when t does a race-free read or write, respectively, which is equivalent to setting $R_x(t)$ or $W_x(t)$, since t's metadata for x is in cache. Thus RADISH's metadata updates are equivalent to those in the vector-clock race detector.

Lock vector clocks are always stored in software, identical to the vector-clock race detector. Thus, with equivalent state and equivalent race checks and metadata updates, RADISH is equivalent to a vector clock race detector, reporting exactly the same races.

A.2 Soundness of In-Hardware Status and Local Permissions

It is straightforward to see that the in-hardware status is preserved via the downgrades on evictions as detailed in Figure 5. Whenever metadata containing a last read or last write of address x is evicted, x's in-hardware status is downgraded to reflect the fact that either the last write or some last reads are not present in hardware. The correctness of local permissions, however, is more involved. The invariant local permissions must preserve is that if an access does not violate local permissions it cannot result in a data race. In other words, if an access is allowed by local permissions, then performing a full race check on that access must succeed. Though this would hold if other threads' permissions were downgraded accordingly whenever one thread performed a memory operation, it is sufficient to downgrade permissions only on coherence events. We characterize a coherence event on data line ℓ initiated because of a memory access a by thread t in a program trace α as a set $e(\ell) = \{a\} \cup \{e_u(\ell) \mid u \neq t\}$ of events in each other thread that occur atomically (and thus consecutively) immediately preceding a in α. For each thread u, we say $e_u(\ell)$ happens after all events a by thread u that precede $e_u(\ell)$ in α and $e_u(\ell)$ happens before all events b by thread u that follow $e_u(\ell)$ in α. We attribute an update of u's permissions for an address x during $e(\ell)$ to $e_u(\ell)$, or to a for thread t. Let ℓ be a data line containing x, let b be a memory operation on x by thread t in program trace α, let d be the last event that precedes – or is – b in α on which t's local permission for x is set, and let $a \neq b$ be an access to x that conflicts with b (at least one of a or b is a write) and precedes b in α. It suffices to show that if d sets permissions that allow b then a must happen before b.

The proof is by contradiction. Suppose d sets permission p for t on x that allows b and a does not happen before b. Then p must be READ or WRITE to allow b. For all vector clocks $v \in \{C_t, R_x, W_x\}$, let v^f denote their values before event f.

If d is a memory operation, then it performs a full check showing $W_x^d \sqsubseteq C_t^d$ and, if d sets WRITE, that $R_x^d \sqsubseteq C_t^d$. If $d = b$, then clearly a must happen before b since it is represented in W_x^d or R_x^d. This is a contradiction. Otherwise,

by program order, d happens before b, and $W_x^d \sqsubseteq C_t^b$ in both cases and $R_x^d \sqsubseteq C_t^b$ in the WRITE case. Since a does not happen before b, then a must follow d in α. Also, at d, t must have ℓ in non-invalid state. If $a = wr_u(x)$ then there must be some coherence event $e(\ell)$ in α between d and a to give u line ℓ in modified state, where $e_t(\ell)$ would downgrade t's permission on x to NONE, so d is not the last event to set t's permission on x before b, which is a contradiction. Otherwise, $a = rd_u(x)$, $b = wr_t(x)$, and d must set WRITE if b is to be allowed. Then there must be some coherence event $e(\ell)$ in α between a and b to give u line ℓ in modified state, where $e_t(\ell)$ would set t's permission based on whether a happens before b. Thus if a does not happen before b, then $e_t(\ell)$ sets t's permission on x to NONE and b violates this permission, which is a contradiction.

If d is a coherence event $e_t(\ell)$, then it must downgrade t's permission on x from WRITE to READ, as set at some prior memory operation f by t, where $W_x^f \sqsubseteq C_t^f$ and $R_x^f \sqsubseteq C_t^f$. The downgrade must be due to a remote read operation, thus R_x^b has changed from R_x^f and possibly from R_x^d if more remote reads g follow the one that generated the coherence event, however the last-write vector clock has not changed as a result of the read. Any difference between W_x^d and W_x^b may only be due to writes by t, otherwise some coherence event would have downgraded t's permission on x to NONE. Writes by t are ordered by program order with b, so in either case, it will be true that $W_x^b \sqsubseteq C_t^b$. For t's READ permission on x to allow b, b must be a read operation, so the race check would pass if we ran it now (at event b), so a must happen before b, which is a contradiction. Thus, in all cases, if d sets permissions that allow b then a must happen before b. Local permissions are sound even when downgraded only on coherence actions.