PARSNIP: Performant Architecture for Race Safety with No Impact on Precision

Yuanfeng Peng
yuanfeng@cis.upenn.edu
University of Pennsylvania

Benjamin P. Wood
benjamin.wood@wellesley.edu
Wellesley College

Joseph Devietti
devietti@cis.upenn.edu
University of Pennsylvania

ABSTRACT

Data race detection is a useful dynamic analysis for multithreaded programs that is a key building block in record-and-replay, enforcing strong consistency models, and detecting concurrency bugs. Existing software race detectors are precise but slow, and hardware support for precise data race detection relies on assumptions like type safety that many programs violate in practice.

We propose Parsnip, a fully precise hardware-supported data race detector. Parsnip exploits new insights into the redundancy of race detection metadata to reduce storage overheads. Parsnip also adopts new race detection metadata encodings that accelerate the common case while preserving soundness and completeness. When bounded hardware resources are exhausted, Parsnip falls back to a software race detector to preserve correctness. Parsnip does not assume that target programs are type safe, and is thus suitable for race detection on arbitrary code.

Our evaluation of Parsnip on several PARSEC benchmarks shows that performance overheads range from negligible to 2.6x, with an average overhead of just 1.5x. Moreover, Parsnip outperforms the state-of-the-art Radish hardware race detector by 4.6x.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;
• **Software and its engineering →** *Software maintenance tools*;

KEYWORDS

multithreaded programming, data race detection, hardware support

ACM Reference Format:
Yuanfeng Peng, Benjamin P. Wood, and Joseph Devietti. 2017. PARSNIP: Performant Architecture for Race Safety with No Impact on Precision. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123946

1 INTRODUCTION

Data race freedom is a critical safety foundation for shared-memory multithreaded programs. Data races may cause unpredictable behavior and factor in many higher-level concurrency errors such as atomicity violations, ordering violations, or determinism violations. Memory consistency models for mainstream multithreaded languages guarantee sequential consistency in data race free programs, but offer weak or undefined semantics for programs with data races [6, 28]. Thus data races may lead to silent violations of sequential consistency and perplexing program behavior.

Researchers have proposed *data race exceptions* to make data races explicit run-time errors in the programming model and simplify semantics of multithreading [2, 12, 25, 30]. Data race exceptions have many programmability benefits but require data race checking support that is ***sound*** (no missed data races), ***complete*** (no false data races), and ***efficient*** enough for production use.

An extensive body of prior work on static, dynamic, and hybrid techniques for general data race checking with software or hardware support has yielded suitable assistants for debugging and testing programs with data races (*e.g.*, [1, 9, 12, 15, 20, 37, 46, 47, 55, 58]). Yet existing approaches remain limited by missing true data races, reporting false data races, or incurring large run-time overheads. To date, implementation proposals focused on data race *exceptions* mainly focus on detecting data races that may violate sequential consistency, or other related memory consistency properties [5, 25, 30, 51]. Yet detecting *all* data races remains an important goal. For example, identifying general data races is a key part of several algorithms for checking or enforcing higher-level properties of multithreaded programs, such as atomicity [17, 33] or determinism [3, 8, 39].

Sound and complete data race detectors implemented in software exhibit performance that is not acceptable in production [12, 15, 16]. The state-of-the-art sound and complete hardware-supported data race detector, Radish [9], generally incurs run-time overheads below 2x. However, this performance is achievable only via two strong assumptions: 1) applications perform only type-safe memory accesses at consistent data granularities and 2) extra cores are available to perform asynchronous checks with relaxed exception delivery. Some C/C++ applications break the type-safety assumption. Relaxed exceptions complicate semantics and make useful exception handling difficult or impossible. Radish is notably less efficient when these unsafe optimizations are disabled. It remains an open question whether efficient, precise data race detection can be provided in more realistic settings.

This paper presents Parsnip, a novel sound and complete data race detector that integrates hardware and software support to achieve good performance by exploiting redundancy in data race detection. Parsnip is based on the state-of-the-art sound and complete software data race detection algorithm FastTrack [15], but accelerates common cases of the analysis by hosting and analyzing frequently used metadata directly in hardware with simple architecture support. Like Radish, the Parsnip hardware falls back on a general software algorithm when finite hardware checking resources are exceeded. In contrast to Radish, the Parsnip design

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10.
https://doi.org/10.1145/3123939.3123946

achieves a significant reduction in performance overheads by exploiting locality and redundancy much more effectively. Parsnip removes metadata redundancy across arbitrary memory locations, and is not subject to the limitations of static analysis like RedCard [16], does not focus solely on arrays as SlimState [56] does, and does not compromise on precision as with [52]. Furthermore, Parsnip does not rely on Radish's simplifying assumptions of type safety and asynchronous checking; Parsnip works for general C/C++ code and does not require any spare cores.

Data race detectors must record and check the logical time of the last access to each memory location. Logical time changes at synchronization events, which are typically much less frequent than memory accesses. The substantial temporal and spatial locality among data accesses translates into high rates of temporal and spatial redundancy in data race checks and metadata. The key contribution of Parsnip is a hybrid hardware-software design that exploits redundancy of data race checks and metadata in two ways. First, Parsnip deduplicates data race detection metadata storage efficiently on the fly by splitting and sharing metadata across multiple locations. Second, Parsnip resolves many checks with only partial metadata and lower storage overhead by memoizing the metadata needed for common-case checks as compact *access capabilities*.

To maintain analysis metadata in hardware, Parsnip splits metadata storage for a given data location between (A) a metadata line shadowing each data line in cache and (B) a small per-core metadata table. Each metadata line stores information about the access history of each location in its shadowed data line in a flexible format, including: (1) what thread performed the last access, (2) the type (read or write) of the last access, and (3) a reference into a per-core hardware table of additional metadata, where each hardware table entry may be referenced by many locations.

Parsnip falls back on the software handler to persist or load additional access history metadata when access capabilities or the metadata table cannot satisfy a check, when the per-core table overflows, and when metadata lines are filled or evicted in the cache. Metadata lines share the standard data cache hierarchy along with the data lines they track, by occupying an unused portion of the physical address space.

This paper makes the following contributions:

- We describe the Parsnip architecture, which provides hardware support for precise data race detection with an average performance overhead of just 1.5x.
- Parsnip outperforms the state-of-the-art Radish race detector by 4.6x if Radish's type safety assumption and extra cores are removed. 3.4x faster than Radish even Radish uses its unchecked assumptions, yet Parsnip does not rely on type safety or require extra cores.
- We leverage cross-location redundancy in race detection metadata for efficient on-chip metadata storage.
- We demonstrate how to encode the common cases of race detection metadata as compact *access capabilities*, without compromising soundness and completeness.

The remainder of this paper is organized as follows: Section 2 reviews canonical algorithms for precise dynamic data race detection. Section 3 introduces the core Parsnip system design, and Section 4 describes key optimizations. Section 5 presents our experimental evaluation. Section 6 reviews related work and Section 7 concludes.

2 DATA RACE DETECTION OVERVIEW

This section gives a brief overview of happens-before race detection, the algorithm underpinning Parsnip and other precise data race detectors. We start by defining a multithreaded program as a single trace of operations, with operations from each thread interleaved. Operations consist of memory reads and writes, lock acquires and releases, and thread fork and join. The *happens-before* relation $\xrightarrow{hb}$ is a partial order over these trace events [22]. Given events a and b, we say a *happens before* b, written $a \xrightarrow{hb} b$, if: (1) a and b are from the same thread and a precedes b in program order; or (2) a precedes b in synchronization order, *e.g.*, a is a lock release $rel_t(m)$ and b the subsequent acquire $acq_u(m)$; or (3) (a, b) is in the transitive closure of program order and synchronization order. If a happens before b then we can equivalently say that b *happens after* a. Two events not ordered by happens-before are said to be *concurrent*. Two memory accesses to the same address form a *data race* if they are concurrent and at least one access is a write.

Vector clocks [14, 18] have been proposed as a data structure to track the happens-before relation at runtime. A vector clock v is a map from thread ID's to logical clocks (integers). Two important operations on vector clocks are *union*: the element-wise maximum of two vector clocks ($v_1 \sqcup v_2 = v_3$ s.t. $\forall t, v_3(t) = \max(v_1(t), v_2(t))$); and *happens-before*: the element-wise comparison of two vector clocks ($v_a \sqsubseteq v_b$ is defined to mean $\forall t, v_a(t) \leq v_b(t)$).

If a trace is race-free, then all writes to an address must be totally ordered. Thus every read and write of x must happen after the last write to x and every write to x must also happen after all last reads of x since the last write to x. Optimized race detectors [12, 15] exploit this property by recording only a single last write per address, and Parsnip operates similarly. The last write information that must be recorded can be thought of as one thread-clock mapping from the vector clock, and is referred to as an *epoch* [15] and is written $t@c$ to represent a write by t at local time c. We generalize the happens-before operation $\sqsubseteq$ to allow comparing a "scalar" epoch with a vector clock; the epoch is conceptually expanded into a vector where all entries are zero except the entry the epoch represents, and then the normal element-wise vector comparison occurs. Much of the time, read operations are also totally ordered which allows the last read to be represented as an epoch as well [15]. We elide this important optimization here for simplicity, but discuss Parsnip's use of it later in Section 3.2.

A vector-clock race detector keeps four kinds of state:

- For each thread t, vector clock C_t represents the last event in each thread that happens before t's current logical time.
- For each lock m, vector clock L_m represents the last event in each thread that happens before the last release of m.
- For each address x, vector clock R_x represents the time of each thread's last read of x *since the last write by any thread*. If thread t has not read x since this write, then $R_x(t) = 0$.
- For each address x, epoch W_x represents the thread that last wrote to x and the time of its write.

Initially, set all L and R vector clocks to v_0, where $\forall t, v_0(t) = 0$. W write epochs are set to e_0, which is equivalent to v_0. Each thread t's initial vector clock is C_t, where each thread increments its own entry in its vector clock, *i.e.*, $C_t(t) = 1$ and $\forall u \neq t, C_t(u) = 0$. This represents the fact that threads are initially executing concurrently with respect to one another.

On a lock acquire $acq_t(m)$, set $C_t := C_t \sqcup L_m$. By acquiring lock m, thread t has synchronized with all events that happened before the last release of m, so t increases its logical clock to be well-ordered with these prior events. On a lock release $rel_t(m)$, set $L_m := C_t$, ordering t with events that happened before this release, then increment t's entry in its own vector clock C_t to ensure that subsequent events in t are marked as concurrent with respect to other threads.

On a read $rd_t(x)$, first check if $W_x \sqsubseteq C_t$. Failure of this check indicates a data race, where t's read of x is concurrent with the most recent write to x. Otherwise, set t's entry in R_x to t's current logical clock, $C_t(t)$. On a write $wr_t(x)$, check that t is well-ordered with respect to the previous write $W_x \sqsubseteq C_t$ and with respect to previous reads $R_x \sqsubseteq C_t$. Failure of either of these checks indicates a data race. If t's write is race-free, clear all last reads by setting R_x to v_0, and set the last write to t's current time $W_x := t@C_t(t)$.

3 THE PARSNIP SYSTEM

The work of a precise dynamic data race detector comprises two components. The detector must: update information about cross-thread ordering on each synchronization event; and check and update per-location access history on each memory access event. Memory accesses typically occur much more frequently than synchronization, so optimizing the analysis, update, and representation of per-location access histories is crucial to data race detector performance.

The guiding principle of the PARSNIP system is to handle the common cases of memory access checks and access history metadata storage in hardware and fall back on a flexible software layer for uncommon cases that require additional access history to maintain soundness and completeness. PARSNIP leverages the following observations about the empirical behavior of dynamic race detection. First, access history metadata for adjacent data locations are likely to be similar or identical, so PARSNIP tries to reduce access history metadata redundancy across memory locations at various levels. Second, most runtime checks need information about only the most recent access, so PARSNIP organizes access history metadata to keep last-access information in a hardware-managed format on chip or in cache, even when full access history for a memory location may be available only in a software-managed format in memory.

The remainder of this section defines the core PARSNIP system in detail. Section 3.1 describes how a simplified version of PARSNIP addresses central design issues. Sections 3.2-3.5 remove these simplifications and describe in detail how key pieces of PARSNIP state are organized. Section 3.6 describes PARSNIP's hardware support, Section 3.7 how runtime checks operate, and Section 3.8 PARSNIP's system-level interactions. Finally, Section 3.9 shows how PARSNIP works on a detailed example trace.

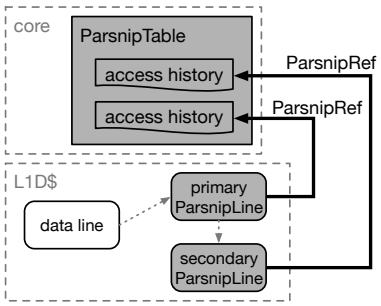

Figure 1: An overview of Parsnip's key pieces of state: ParsnipLines, ParsnipRefs and the per-core ParsnipTable.

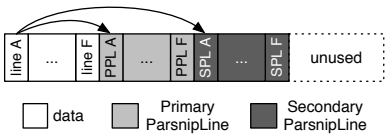

Figure 2: Parsnip's physical address space layout.

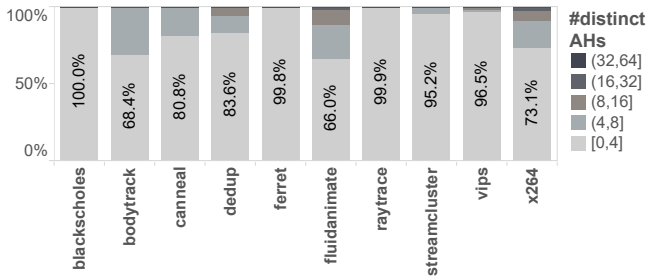

Figure 3: Distribution of the number of distinct access histories per 64B cache line.

3.1 Simplified PARSNIP

Here we outline the operation of a simplified version of PARSNIP, and in subsequent sections we discuss how this simple scheme is extended to improve performance and retain precision.

The first task a race detector faces when needing to check an access to location x is to find the associated metadata for x. PARSNIP steals 2 bits from the physical address space to afford a simple data:metadata mapping based on physical addresses (Figure 2), constructed to map x and its metadata to different cache sets to reduce conflict misses. We describe interactions with address translation in Section 3.8. PARSNIP manages metadata at cache line granularity, so for each cache line of program data, there is a corresponding ParsnipLine given by this mapping. These ParsnipLines reside in the data cache hierarchy and compete for space with regular program data (as shown in Figure 1; Primary versus Secondary ParsnipLines are explained later in Section 3.2).

While the most straightforward use for a ParsnipLine would be to record an access history directly, we find that doing so is not space efficient. We counted how many unique access histories appear across ParsnipLines for our workloads, and show the results in

Access History Type	Read Check requires:	Write Check requires:
last access	last access	last access
RAW	last read + last write, or last access	last access
RsAW	last read + last write, or last access	all reads
Rs	last access	all reads

Table 1: Components required for different types of access histories and memory access checks.

Figure 3. While the 64B cache lines in our system can theoretically require a distinct access history for each byte, on every program (except fluidanimate and x264) the number of distinct access histories per line is *never* more than 16. Even for fluidanimate and x264, fewer than 2% of lines have more than 16 distinct access histories. This result is unsurprising given that programs typically access data at multi-byte granularity, and exhibit spatial locality. These properties make it highly likely that nearby locations will have the same access history.

To take advantage of the redundancy among access histories, a ParsnipLine contains a collection of references (called ParsnipRefs) which point to entries in a per-core ParsnipTable. The ParsnipTable entries themselves contain the access history information. By adding a level of indirection, Parsnip is able to share ParsnipTable entries across a large number of program locations. Because only a small number of ParsnipTable entries are typically needed, each ParsnipRef can itself be small, helping to reduce Parsnip's footprint in the data cache.

However, several challenges remain. Access histories are of variable size, and at their largest can require tracking a clock value for each thread in the program – much larger than can fit into a fixed-size hardware table. The ParsnipLine encoding must similarly be able to exploit the common case while supporting precise tracking even when there are 64 unique histories in a single line. We address these challenges in the following sections.

3.2 Access History Organization

To encode access histories of memory locations efficiently, Parsnip exploits the observation that most data race checks need only a small part of the full access history.

Access histories of memory locations can be classified into the following types: 1) *last access*, which contains only the most recent write/read; 2) *RAW*, which consists of the last write and the last read that happens-after the write; 3) *RsAW*, which keeps the last write and all concurrent reads that happen-after the write; and 4) *Rs*, which tracks all concurrent reads with no prior write.[1]

Table 1 shows what piece(s) of the access history a runtime read or write check needs. In most cases a check can be done with only the last access, or both the last read and the last write; only when a write occurs after some concurrent reads (where the access history is in RsAW/Rs form) does the check require all concurrent reads since the last write. In fact, such cases are rare in practice, which implies that an efficient race detector can complete most checks with a history of at most two accesses, without losing soundness.

[1] Well-defined C/C++ programs should precede any read to a memory location by an initializing write, but some programs (including our benchmarks) read uninitialized data in practice. Another potential source of reads before writes is the use of facilities such as demand-zeroed pages that initialize memory contents with other mechanisms.

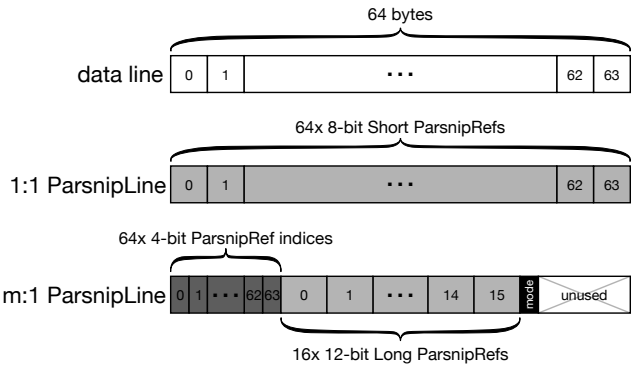

Figure 4: A ParsnipLine in 1:1 mode (middle) and m:1 mode (bottom)

Based on this observation, Parsnip organizes the access history in a way such that the last read and last write information is in hardware in most cases, with the rest of the access history stored in software. As discussed in Section 3.1, each data location x has a corresponding Primary ParsnipRef that points to its access history in the ParsnipTable. To maintain a fixed size, each ParsnipTable entry contains a single clock value (and a reference count, explained later in Section 3.5). Thus, tracking the last read *and* last write for x requires two ParsnipRefs: the **Primary** ParsnipRef and the **Secondary** ParsnipRef. These primary and secondary references are stored in Primary and Secondary ParsnipLines, accordingly, and their locations are computed from the corresponding data address (Figure 2).

The Primary ParsnipRef refers to the last access, which is a read for the RAW and RsAW access history types. A Primary ParsnipRef contains several fields (detailed in Section 3.4), including a *hasNext* bit indicating whether x has a Secondary ParsnipRef. The Secondary ParsnipRef contains the last write when the access history is of RAW or RsAW type. The Secondary ParsnipRef has the same structure as the Primary ParsnipRef, but its *hasNext* bit indicates whether additional access history information, *i.e.*, additional concurrent readers, is stored in software.

The access history of a data location x is thus stored across up to 3 parts: Primary ParsnipRef, Secondary ParsnipRef, and remaining information in software. This organization helps minimize cache pollution, as most race checks can be discharged with the Primary ParsnipRef alone.

3.3 ParsnipLine Format

The results shown in Figure 3 indicate an opportunity for Parsnip to have ParsnipRefs that are larger than a byte while using a single ParsnipLine to track all the ParsnipRefs for a cache line of data. Specifically, ParsnipLines in Parsnip have two modes: 1:1 mode and m:1 mode. Figure 4 shows the two ParsnipLine formats. In 1:1 mode, a ParsnipLine is filled with 8-bit ParsnipRefs, each associated

in a 1:1 fashion with the bytes in the corresponding data line. In m:1 mode, each ParsnipRef is 12 bits in size, and we introduce another layer of indirection to map data line bytes through a 64-entry bitmap to one of the 16 ParsnipRefs. For the i^{th} byte of a data line, its ParsnipRef can be found by first looking up the bitmap to get an index $bitmap[i]$, then reading the $bitmap[i]^{th}$ ParsnipRef. The bitmap occupies 32B, and the ParsnipRefs 24B, with 8B left over. The byte following the ParsnipRefs is used to indicate whether a ParsnipLine is in m:1 or 1:1 mode by reserving a special bit pattern. (detailed in Section 3.4). The other 7B are unused.

Having smaller ParsnipRefs in 1:1 mode means that for each byte less information of the access history can be encoded than the m:1 mode, but in practice even these 8-bit ParsnipRefs is sufficient for many access checks(see Figure 10 in Section 5). The number of distinct ParsnipRefs per ParsnipLine is computed by dedicated hardware (see Figure 6). Whenever a ParsnipRef in m:1 mode needs to switch to 1:1 mode or vice versa, the ParsnipLine logic triggers a rewrite of the ParsnipLine. In practice, 1:1 mode ParsnipLines are rarely needed and mode switches are rare.

3.4 ParsnipRef Format

As described in Section 3.3, Parsnip can have two modes of ParsnipLines. In m:1 ParsnipLines, ParsnipRefs occupy 12 bits, whereas in 1:1 ParsnipLines, ParsnipRefs occupy 8 bits. For brevity, we refer to the 12-bit ParsnipRefs as **Long** ParsnipRefs, and the 8-bit ParsnipRefs as **Short** ParsnipRefs. Figure 5 shows the format of both Long and Short ParsnipRefs. Each ParsnipRef starts with a 6-bit *tid* that stores the thread that performed the access, followed by a R/W bit indicating whether the ParsnipRef points to a read or write access, and a *hasNext* bit (introduced in Section 3.2) indicating whether additional access history information is available in a Secondary ParsnipRef or in software. For brevity, we define the pair $(tid, r/w)$ to be an **access capability**; therefore, a Short ParsnipRef encodes an access capability. In a Long ParsnipRef, another 4 bits are used to hold an index into the 16-entry ParsnipTable, whose entries contain clock values. Together, the *tid* field and ParsnipTable entry constitute an epoch which can be used to perform race detection checks.

As mentioned in Section 3.3, Parsnip reserves a special bit pattern to distinguish m:1 mode ParsnipLines from 1:1 lines. Specifically, the byte value 0xFF is reserved to mean m:1 mode, whereas all other values of the *mode* byte are interpreted as a Short ParsnipRef, indicating that the current ParsnipLine is in 1:1 mode. One implication is that any Short ParsnipLine must not have the *tid* field set to be 0b111111 to avoid collision with the special mode byte pattern.

3.5 ParsnipTables

ParsnipTables are small per-core tables that hold logical times. Each ParsnipTable entry consists of a 64-bit clock and a 24-bit reference count to record ParsnipRefs that refer to this entry. Parsnip recycles an entry once its reference count reaches 0. The value i of the 4-bit *index* field in a Long ParsnipRef points to the i^{th} entry in the ParsnipTable of thread t, where t is the value of the ParsnipRef's *tid* field. Parsnip reserves the value 0xF to indicate an "invalid" ParsnipTable index, which is used to represent clock values that are not present in the ParsnipTable (and must be retrieved from software instead). Given the 4-bit indices in a Long ParsnipRef, each ParsnipTable can hold up to 15 distinct clock values. A 15-entry ParsnipTable occupies just 165B of space.

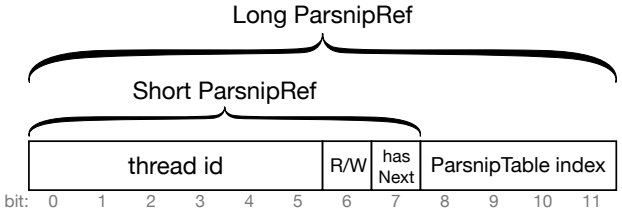

Figure 5: Format of Short and Long ParsnipRefs.

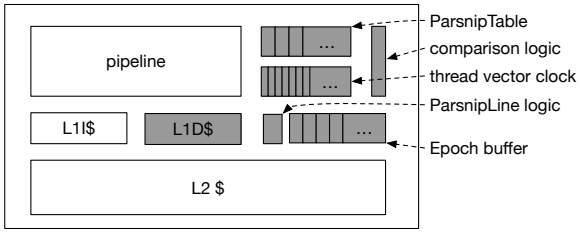

Figure 6: Parsnip additions (shaded) to a conventional core.

Although quite small in size, ParsnipTables allow a large proportion of the runtime checks to be completed in hardware. Whenever an epoch $c@t$ of an access to a memory location x needs to be recorded, Parsnip first traverses the ParsnipTable of thread t looking for an entry that already holds the clock value c. If such an entry is found, the reference count of the entry is incremented and the index of the entry is written in the corresponding ParsnipRef for x. Otherwise, a new ParsnipTable entry needs to be allocated to hold c. If there is free space in the ParsnipTable, then the index of the newly allocated entry is written in the ParsnipRef of x. Otherwise, the ParsnipTable is full and a ParsnipTable *rejection* occurs: the invalid index 0xF is written in the ParsnipRef, and the epoch value is persisted into software.

The reference count in a ParsnipTable entry tracks the number of ParsnipRefs in cache that point to that entry. Therefore, if a ParsnipRef leaves the data cache, the corresponding reference count is decremented. To implement this, an eviction handler is triggered when a ParsnipLine is evicted from the last-level data cache; inside this eviction handler, all reference counters corresponding to the indices of the ParsnipRefs being evicted are decremented.

Parsnip adopts the above ParsnipTable rejection and reference count policy as a simpler alternative to conventional preemptive eviction of least recently used ParsnipTable entries. LRU eviction of table entries is impractical because it requires changes to all ParsnipRefs referencing the victim entry, an unbounded set. Section 4.3 discusses refinements to the ParsnipTable management policy to improve utilization.

3.6 Parsnip Hardware Support

Figure 6 shows an overview of a Parsnip processor core, with shaded blocks showing the components Parsnip adds to a conventional design. As per-thread vector clocks are necessary in most runtime checks, Parsnip stores them on-chip. Parsnip also adds simple logic to compare a component of the per-thread vector clock with an epoch from a location's access history, and a small ParsnipTable to remove redundancy among the access histories of different memory locations (Section 3.5). Parsnip also adds custom logic to track the number of distinct last accesses in a ParsnipLine, to determine whether the line can be encoded in m:1 or 1:1 format (Section 3.3). Due to the limited capacity of ParsnipTables, some access histories may need to be persisted into software. To reduce the latency incurred for these updates, Parsnip adds a hardware structure to buffer these updates (see Section 4.1). Parsnip (like Radish) adds an extra read port to the L1D$ to read data and metadata in parallel.

3.7 Parsnip Access Checks

When a program accesses a location x, Parsnip first looks up x's Primary ParsnipRef in the data cache; if the Primary ParsnipLine is not in cache, a software handler will be invoked to complete the current check, after which the ParsnipLine of x will be updated. If the Primary ParsnipLine is in cache, the ParsnipLine logic decodes it and gets the Primary ParsnipRef of x. Depending on the type of the current access and the content of the Primary ParsnipRef, Parsnip can decide whether all necessary information for the current check has been collected. If so, the check is complete, after which the Primary ParsnipRef of x is updated. Otherwise, Parsnip continues to look up the Secondary ParsnipRef of x. If the Secondary ParsnipLine is not in cache, or the Secondary ParsnipRef still does not provide all necessary information required by the current check, a software handler will be invoked to handle the check and update the ParsnipRefs.

3.8 System-Level Considerations and ISA

The address of a ParsnipLine is derived from the physical address of the corresponding data. Because a ParsnipLine is the same size as a data cache line, cache indexing can occur in parallel with data address translation as with a conventional cache. Once the virtual tag of the data address has been translated to a physical tag by the TLB, the corresponding tag for the associated ParsnipLine can be computed directly. Parsnip does not require a separate TLB access to obtain the tag for the ParsnipLine address.

Because ParsnipLines are physically-addressed, they do not need to be saved and restored on a context switch. Conventional address space isolation between processes keeps ParsnipLines isolated as well. Context switches must, however, save and restore ParsnipTable contents and per-core vector clocks.

Parsnip's ISA support consists primarily of checked load and store instructions that trigger a race check when executed, similar to [57]. This allows application code to interleave at fine granularity with code for a runtime system or trusted library, and allows users to easily opt-in to Parsnip support. Parsnip additionally requires the following work to be done in software: 1) handling ParsnipLine evictions from cache, 2) doing race checks in software when the hardware has insufficient information, and 3) updating the per-thread vector clocks on synchronization operations. User-level interrupts are used to quickly transfer control to software when these events occur.

An eviction handler is invoked when a ParsnipLine l leaves the cache hierarchy. The software handler must determine l's owning process (which may be de-scheduled). This is done with OS support by looking up l's corresponding data physical address in the OS "reverse frame map" which maps from physical pages to virtual pages. For every Long ParsnipRef containing a valid index *idx* into thread t's ParsnipTable, the handler gets the clock value c from thread t's ParsnipTable, decrements the entry's reference counter, and saves the epoch $c@t$ into a software representation. For Short ParsnipRefs, or Long ParsnipRefs with an invalid ParsnipTable index, the access history in software is already up-to-date and no further action needs to be taken.

A software check handler is called when the hardware has insufficient information for a runtime check (*e.g.*, the Primary ParsnipRef is not cached, or a write needs to check against the complete set of concurrent readers). The software check handler is invoked with the partial access histories from hardware (if available from the ParsnipTable), which, combined with the information stored in software, forms the complete access history of the memory location being checked. After finishing the current check, this handler also updates the Primary ParsnipRef and, if necessary, the Secondary ParsnipRef, in accordance with the updated access history.

In order for Parsnip to track synchronization events across threads, the synchronization library must be modified such that Parsnip's per-thread vector clocks are updated properly during each synchronization operation.

3.9 Parsnip Example Trace

Table 2 gives an example trace illustrating how Parsnip works for a memory location x. The first column indexes the trace. The next three columns show operations by three threads. The symbol "!" marks accesses on which a data race is reported. Remaining columns show status of analysis metadata *after* the corresponding operation is completed. Next is the Parsnip access history, including ParsnipRefs, a partial ParsnipTable, and the access history stored by the software layer. The final columns show vector clocks tracking synchronization order as in Section 2. Vector clocks are notated as sets of epochs. We conflate cores with threads. Additional notation is introduced below.

At step 1, t_1 writes x. No primary ParsnipRef exists for x, so Parsnip invokes a software handler. The software handler finds that there are no prior accesses to x in software access history, so the check succeeds. Free entry 0 from t_1's ParsnipTable is allocated to store the clock at which this access occurs (1, as obtained from the thread VC for t_1). ParsnipTable entries take the form $index \mapsto (clock, ref\ count)$. We show only some entries for each ParsnipTable in Table 2. A ParsnipRef t_1, W, F, [0] is initialized for x, showing that the last access was a write (W) by thread t_1. There is no secondary ParsnipRef (F). The ParsnipRef contains an index [0] into t_1's ParsnipTable, where the local clock of the access is recorded.

Execution Trace				Parsnip Access History for x							Synchronization History			
	Threads			ParsnipRefs		ParsnipTables (partial)			SW Metadata		C: Thread VCs			L: Lock VCs
Step	t_1	t_2	t_3	Primary	Secondary	t_1	t_2	t_3	W_x	R_x	t_1	t_2	t_3	l_1
0								$4 \mapsto (16,1)$			$\{1@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	
1	wr x			$t_1, \mathsf{W}, \mathsf{F}, [0]$		$0 \mapsto (1,1)$		$4 \mapsto (16,1)$			$\{1@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	
2	rd x			$t_1, \mathsf{W}, \mathsf{F}, [0]$		$0 \mapsto (1,1)$		$4 \mapsto (16,1)$			$\{1@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	
3	rel l_1			$t_1, \mathsf{W}, \mathsf{F}, [0]$		$0 \mapsto (1,1)$		$4 \mapsto (16,1)$			$\{2@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	$\{1@t_1\}$
4	rd x			$t_1, \mathsf{R}, \mathsf{T}, [1]$	$t_1, \mathsf{W}, \mathsf{F}, [0]$	$0 \mapsto (1,1)$ $1 \mapsto (2,1)$		$4 \mapsto (16,1)$			$\{2@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	$\{1@t_1\}$
5		acq l_1		$t_1, \mathsf{R}, \mathsf{T}, [1]$	$t_1, \mathsf{W}, \mathsf{F}, [0]$	$0 \mapsto (1,1)$ $1 \mapsto (2,1)$		$4 \mapsto (16,1)$			$\{2@t_1\}$	$\{1@t_1, 1@t_2\}$	$\{16@t_3\}$	$\{1@t_1\}$
6		rd x		$t_2, \mathsf{R}, \mathsf{T}, [1]$	$t_1, \mathsf{W}, \mathsf{T}, [0]$	$0 \mapsto (1,1)$	$1 \mapsto (1,1)$	$4 \mapsto (16,1)$		$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 1@t_2\}$	$\{16@t_3\}$	$\{1@t_1\}$
7		rel l_1		$t_2, \mathsf{R}, \mathsf{T}, [1]$	$t_1, \mathsf{W}, \mathsf{T}, [0]$	$0 \mapsto (1,1)$	$1 \mapsto (1,1)$			$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{16@t_3\}$	$\{1@t_1, 1@t_2\}$
8			acq l_1	$t_2, \mathsf{R}, \mathsf{T}, [1]$	$t_1, \mathsf{W}, \mathsf{T}, [0]$	$0 \mapsto (1,1)$	$1 \mapsto (1,1)$	$4 \mapsto (16,1)$		$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 16@t_3\}$	$\{1@t_1, 1@t_2\}$
9			! wr x	$t_3, \mathsf{W}, \mathsf{F}, [4]$	—	—	—	$4 \mapsto (16,2)$		$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 16@t_3\}$	$\{1@t_1, 1@t_2\}$
10			rel l_1	$t_3, \mathsf{W}, \mathsf{F}, [4]$	—	—	—	$4 \mapsto (16,2)$		$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 17@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$
11			rd x	$t_3, \mathsf{R}, \mathsf{T}, \perp$	$t_3, \mathsf{W}, \mathsf{F}, [4]$	—	—	$4 \mapsto (16,2)$		$\{17@t_3\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 17@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$
12			rel l_2	$t_3, \mathsf{R}, \mathsf{T}, \perp$	$t_3, \mathsf{W}, \mathsf{F}, [4]$	—	—	$4 \mapsto (16,2)$		$\{17@t_3\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 18@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$
13			wr x	$t_3, \mathsf{W}, \mathsf{F}, \perp$	—	—	—	$4 \mapsto (16,1)$	$18@t_3$	—	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 18@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$
14		! rd x		...	...	...	...	...	...	...	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 18@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$

Table 2: An example trace showing how Parsnip checks accesses to a memory location x.

Lock operations in steps 3 and 5 induce happens-before ordering. Vector clock updates follow Section 2. Since the lock release increments t_1's epoch to $2@t_1$, the read at step 4 is in an epoch different from the epoch encoded in x's primary ParsnipRef, so it needs to be recorded as the last access to x. Since this access is a read, information about the prior write must be retained. The ParsnipRef for the past access becomes the secondary ParsnipRef and a fresh primary ParsnipRef is recorded, with a freshly allocated entry [1] in the ParsnipTable, where the current clock (2) is stored.

The next read of x at step 6 is done by thread t_2, concurrently with the last read encoded in the primary ParsnipRef. To check for a potential write-read race, Parsnip also checks the secondary ParsnipRef. Since there is space for information about only 2 accesses in hardware, the older read by t_1 is saved to software and the read by t_2 is recorded as a fresh primary ParsnipRef, using a newly allocated entry [0] in t_2's ParsnipTable. The target ParsnipTable is determined by the thread in the ParsnipRef.

Next, thread t_2 releases lock l_1 (step 7) and t_3 acquires it (step 8), establishing happens-before order. At step 9, the write by thread t_3 checks against both ParsnipRefs, but the secondary ParsnipRef indicates with its *hasNext* bit (T) that there is additional history in software. A software check is invoked and finds a data race with the read at $2@t_1$ (step 4). For illustration purposes, we assume Parsnip continues past this data race report and records this access as usual. Since it is a write, all access history about x is obsolete, and it returns to a single primary ParsnipRef recording the write in epoch $16@t_3$, using entry [4] in t_3's ParsnipTable, which currently holds clock 16. Note that the software history retains information about an older read. Parsnip could preemptively erase this, but there is no need since the new primary ParsnipRef shows with its *hasNext* bit (T) that there is neither a secondary ParsnipRef nor history in software, so future accesses will never inspect that history (even though doing so would not affect precision).

Thread t_3 then releases lock l_1 at step 10, incrementing its logical clock to 17. Thread t_3 reads x at step 11 in the same thread as the last access indicated by the primary ParsnipRef, but in a new epoch. However, the ParsnipTable of t_3 is still full (not shown) and a 17 entry is not available, so the clock value 17 cannot be stored. Parsnip records a fresh primary ParsnipRef for x, assigned to t_3, but with the invalid index (notated $\perp$). Parsnip also saves the full epoch value $17@t_3$ to software. Since the existing primary ParsnipRef indicated that there was neither a secondary ParsnipRef nor history in software, the software overwrites outdated history.

Lock l_2 is released by thread t_3 at step 12 (Table 2 does not show metadata for l_2 in L), followed by a write to x in t_3 at step 13. The check on this write completes after decoding the primary ParsnipRef alone. Even though the primary ParsnipRef indicates there is a secondary ParsnipRef, the current write is in the same thread as the last read encoded by the primary ParsnipRef, so the current write is race-free. Since t_3's ParsnipTable is still full (not shown), the full epoch value $18@t_3$ is persisted into software. As this is a write, all software read history is now stale and is erased. Finally, when t_2 reads x at step 14, the primary ParsnipRef lacks a table index, so Parsnip performs a software check to check the last write epoch, finding a write-read race.

4 OPTIMIZATIONS

This section describes three key optimizations to reduce dependence on the high latency Parsnip software layer in cases where on-chip resources are insufficient to recover or record necessary access history. These optimizations avoid unnecessary software interactions by buffering the transfer of evicted access history metadata to software (Section 4.1); prefetching access history information from software (Section 4.2); and varying the policy for eviction of ParsnipLines from cache (Section 4.3).

4.1 Buffering Evictions of Access History to Software

Parsnip must persist access history information to software to preserve soundness in two cases that are neither common nor rare enough to ignore the high latency of the software layer: (1) on ParsnipTable rejections (Section 3.5) and (2) when recording a new read epoch in a RsAW-mode access history, where Parsnip stores only the last read and last write in hardware and maintains information about other reads in software (Section 3.2).

To reduce the latency of persisting epochs to software, we extend the basic Parsnip design to add a 32-entry coalescing *epoch store buffer* to each core (Figure 6). Each entry holds the address of the accessed data location, the memory access type (read/write), and the epoch in which the access occurred. Parsnip buffers an entry when it needs to persist access history to software. The latency of inserting an entry into the buffer can be hidden by the cost of the corresponding ParsnipLine update. When the buffer fills, a software handler is invoked to drain all buffered entries to software.

When insufficient access history is available on-chip in ParsnipLines or ParsnipTables, Parsnip queries the software layer only if it cannot find the required access history by snooping the epoch store buffer. The buffer also coalesces entries for the same address and access type. For example, if the buffer contains an entry for a read to data location x in epoch e_1 and a read to x in newer epoch e_2 must be persisted to software, the entries are coalesced, preserving only the most recent read to x in e_2. Upon last-level cache misses on a ParsnipLine, the software handler also snoops the epoch store buffers to ensure that any pending updates to x's access history is visible to later accesses.

4.2 Access History Prefetching and Prediction

When an access check by thread t for data location x finds insufficient in-hardware access history, Parsnip invokes a software handler to resolve the check for location x from the full software-managed access history. To exploit spatial locality, Parsnip additionally fills the ParsnipLine for x, setting access capabilities for all other locations that are tracked by the same ParsnipLine.

Rather than simply filling ParsnipRefs for other locations $y \neq x$ in the same ParsnipLine as x's ParsnipRef based on access history, the software handler *speculatively checks a future access* to y of the same type (read/write) and by the same thread t as the access that triggered the software check of y's neighbor, x. If the speculative check determines that such an access to y would be data race free according to the current access history, it fills a ParsnipRef for y with the access capability component set to describe this predicted future access, but leaves the ParsnipTable index component of the ParsnipRef *invalid*.

This optimization preserves soundness and completeness if the next access to y is predicted correctly, as the predicted future access check will resolve based on the ParsnipRef access capability for y and record the up-to-date epoch of this new access, achieving the same result as if it had dispatched the check to software.

If a speculative check mispredicts the next access to y and a different thread $u \neq t$ makes the next access to y, soundness and completeness are still preserved. An access to y by thread u will find the ParsnipRef access capability assigned to t with an invalid ParsnipTable index, thus triggering a search through the epoch store buffer (Section 4.1) or the software layer to find the most up-to-date access history for y. Since speculative checks update only the access capability of a ParsnipRef, no record of the predicted access is found in either the epoch store buffers or software. Parsnip will thus complete a full access check against the same access history it would have checked without speculation. In sum, Parsnip remains sound and complete with this optimization.

4.3 ParsnipTable and ParsnipLine Management

To ensure effective use of limited table space, ParsnipTable management must exploit locality and favor entries that are likely to be reused soon. However, the simple policy of reference counts and rejections established in Section 3.5 has limitations. ParsnipRefs for old, infrequently-used data that remain in the last-level cache may pin ParsnipTable entries with nonzero reference counts, leaving no table space for newer, frequently-used epochs, creating high rates of ParsnipTable rejections and expensive software checks.

A simple alternative is to invalidate and persist a ParsnipLine to software as it leaves the L1 cache. ParsnipRefs for memory locations not accessed recently thus tend to be evicted from L1, decrementing reference counts and freeing more ParsnipTable entries to represent new epochs. However, even hot data locations in programs with large working sets experience frequent evictions and refills to the L1 or even L2. The associated ParsnipLines then suffer frequent round trips to and from to the software layer.

To balance ParsnipTable utilization and software costs, Parsnip adaptively selects an L1 or LLC residence policy for ParsnipLines by tracking the rate of software checks in a window of recent events. Under policies that allow ParsnipLines to reside in the LLC, the heuristic responds to software checks due to excessive ParsnipTable rejections when old ParsnipRefs in the last-level cache pin too many useless ParsnipTable entries. Under L1-only ParsnipLine residence, the heuristic responds to software checks for repeatedly invalidated and refilled ParsnipLines in L1. A user may also mandate a cache residence policy based on profiling or experience. Extending Parsnip to support other heuristics is a promising avenue for future work.

5 EVALUATION

To evaluate Parsnip's performance, we implemented a simulator using Intel PIN [27] to model a 16-core system with MESI coherence protocol, with a realistic memory hierarchy common in commodity processors. Cache lines are 64 bytes. Each core has an 8-way 32KB L1 cache and an 8-way 256KB L2 cache; all cores share a 16-way 32MB L3 cache. Latency of L1, local L2, remote L2, L3, and main memory accesses are 1, 10, 15, 35, and 120 cycles, respectively. The cache subsystem of the Parsnip simulator is implemented by extending the cache implementation in ZSim [45].

Each per-core ParsnipTable has 15 entries (165 bytes total), with each entry consisting of a 64-bit clock and a 24-bit reference count. Each per-core epoch store buffer has 32 entries (544 bytes total), with each entry holding a 64-bit address, a 64-bit clock, and an extra byte encoding both a tid and a r/w bit. Lookups in the local core's ParsnipTable cost 1 cycle; requesting an entry from a remote core's ParsnipTable takes 10 cycles. Latency of epoch store buffer insertions is hidden by the cycles of updating the corresponding ParsnipLine. Accesses to locations on the stack are assumed to be thread-local, and no data race checks are done for stack locations. Software checks and epoch-buffer drains cost 500 cycles plus the cost of any cache accesses triggered.

5.1 Experimental Setup

We evaluate Parsnip with 10 programs from the PARSEC 3.0 benchmark suite [4], including blackscholes, bodytrack, canneal, dedup,

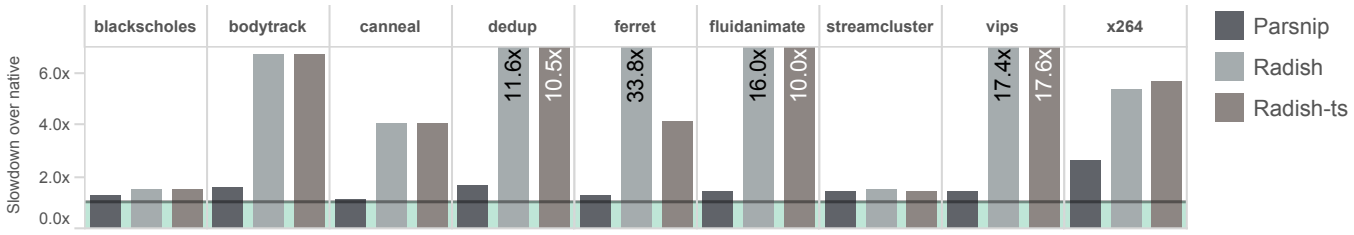

Figure 7: Slowdown of Parsnip, Radish, and Radish-ts, normalized to a system without data race detection support.

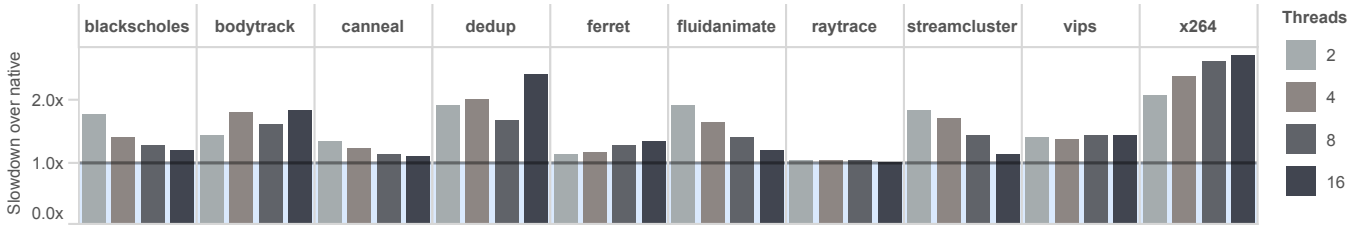

Figure 8: Slowdown of Parsnip with 2, 4, 8 and 16 threads, using the simmedium input.

ferret, fluidanimate, raytrace, streamcluser, vips and x264. We omit the benchmarks facesim, which forks child processes as parallel workers, and swaptions, which has high variance across runs in our experiments. We report performance as the mean of 3 runs.

5.2 Performance

This section compares the performance of Parsnip with Radish [9], the most related work. Figure 7 shows the CPIs of Parsnip and Radish, normalized to the native executions of the respective simulator without data race detection extensions. Results shown here were collected when running with 8 threads and the simsmall input size.[2] We exclude raytrace from these results, since it did not run correctly on the Radish simulator in our experiments. We ran Radish with its relaxed asynchronous checking and unsafe type-safety optimization disabled [9] for the most direct comparison to Parsnip, which provides synchronous data race checks and does not make assumptions about type safety. Additionally, we ran Radish with its type-safety assumption enabled, represented by the Radish-ts bars in Figure 7. This enables better performance but is unsound and incomplete in the presence of type-safety violations.

On all 9 PARSEC benchmarks on which we compare Parsnip against Radish, Parsnip runs faster than Radish. On average (geomean), Parsnip slows the baseline CPI by 1.5x, whereas Radish's average slowdown is 6.9x. The maximum slowdown caused by Parsnip is 2.6x on x264. Radish slows ferret by 33.8x. Parsnip is on average 4.6x faster than Radish, and runs ferret 27.2x faster than Radish. Enabling Radish's type-safety assumption in Radish-ts yields a 1.3x boost over the safe version of Radish. Nonetheless, the optimized Radish-ts remains 3.4x slower than Parsnip on average, and runs vips 12.3x slower than Parsnip does.

[2]Long running times and high memory usage of the Radish simulator made performance experiments for larger input sizes or thread counts infeasible on the Radish simulator in our experiments.

5.3 Scalability

To evaluate Parsnip's performance scaling over the number of threads of the target program, we ran Parsnip experiments with 2, 4, 8 and 16 threads on 16 simulated cores with the simmedium input size. Slowdown of Parsnip's CPI with respect to CPI of the baseline system is shown in Figure 8. On 4 benchmarks (blackscholes, canneal, fluidanimate, streamcluster), Parsnip's overhead decreases with more threads. On 4 additional other benchmarks (bodytrack, dedup, ferret, x264), Parsnip overhead increases with more threads. Parsnip's overhead shows no significant difference on the remaining 2 benchmarks (raytrace and vips). For raytrace, Parsnip shows no appreciable slowdown at all. Parsnip's worst slowdown is 2.7x on x264 with 16 threads.

5.4 Effectiveness of Optimizations

This subsection presents results of several experiments conducted to evaluate the effectiveness of each optimization described in Section 4. Figure 9 shows the slowdown in CPI of several variants of Parsnip, as normalized to the standard Parsnip configuration. Epoch store buffers (described in Section 4.1) have the largest impact. Parsnip runs on average 3.7x slower with epoch store buffers disabled ("no epoch buffers"). Parsnip with access history prefetching and prediction (Section 4.2) offers an average 10% performance improvement versus without ("no AH speculation"). The baseline adaptive selection of the ParsnipLine cache residence policy (Section 4.3) shows comparable average performance with both the fixed L1 and LLC ParsnipLine residence policies ("ParsnipLines in L1, LLC"). However, the adaptive policy avoids the larger slowdowns of the L1 policy on bodytrack, and the LLC policy on blackscholes. In summary, all benchmarks see (often substantial) performance improvement from at least one of the optimizations described in Section 4.

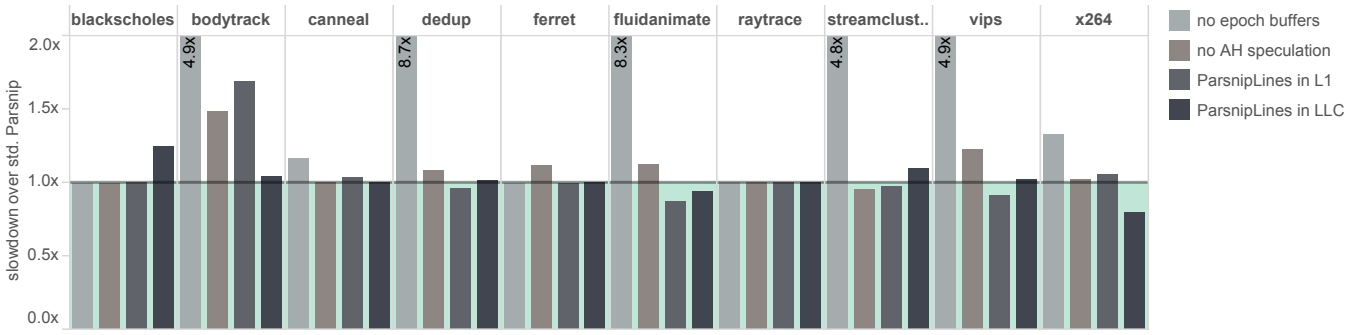

Figure 9: Slowdown of Parsnip with different optimizations disabled.

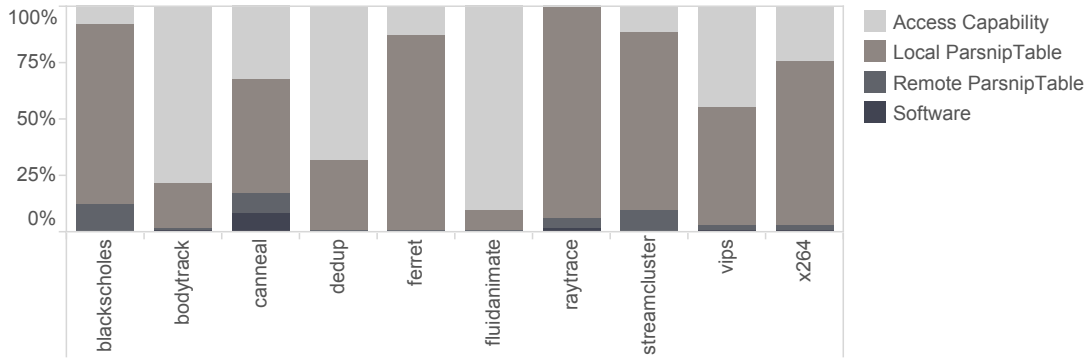

Figure 10: Breakdown of how often each access history source resolves a data race check.

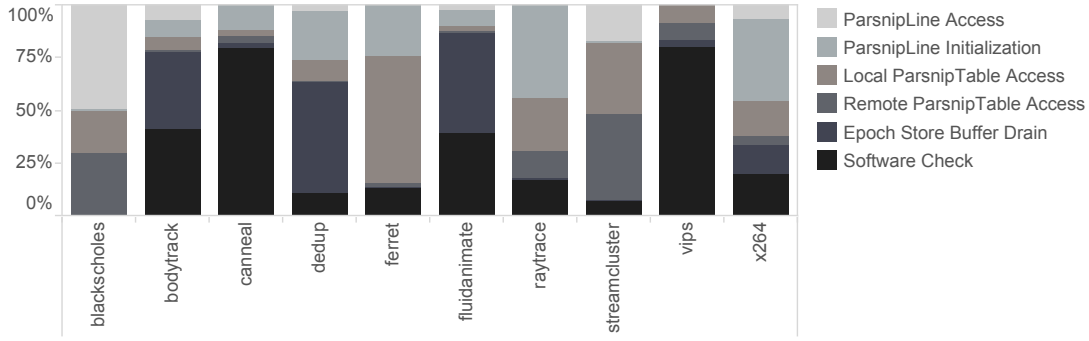

Figure 11: Contribution of architectural events to overall Parsnip overhead.

5.5 Architectural Characterization

We ran several characterization experiments to better understand the sources of Parsnip's overhead.

Figure 10 shows the percentage of data race checks that are resolved by each Parsnip mechanism. ParsnipRefs and the local core's ParsnipTable suffice to resolve at least 84% of checks in each benchmark. Access capabilities in ParsnipRefs alone resolve 68-90% of checks in bodytrack, dedup, and fluidanimate. Remote ParsnipTable lookups resolve a modest number of checks, up to 12% in blackscholes. With the exception of canneal, which requires software checks on 8% of accesses, all other benchmarks require software checks on <1% of accesses, with most <0.6%.

Figure 11 shows the breakdown of Parsnip's overhead from 6 different sources: (a) the cost of accessing ParsnipRefs in cache; (b) the cost of software handling when a ParsnipRef is not in cache; (c) the cost of runtime checks that complete after comparing with a clock in the local ParsnipTable; (d) the cost of runtime checks that finish after comparing with a clock in some remote ParsnipTable; (e) the cost of software check handlers; (f) the cost of saving the contents of an epoch buffer to software.

Benchmark	Epoch buffer drains	Parsnip-Line evictions	Parsnip-Table rejections	SW checks
blackscholes	0.00	0.00	0.00	0.01
bodytrack	0.00	0.00	0.13	0.91
canneal	0.00	0.00	0.03	2.57
dedup	0.01	0.00	0.49	0.57
ferret	0.00	0.00	0.00	0.17
fluidanimate	1.38	0.00	79.15	0.34
raytrace	0.00	0.00	0.00	0.02
streamcluster	0.00	0.00	0.01	0.05
vips	0.00	0.00	0.02	0.45
x264	0.00	0.00	0.00	1.45

Table 3: Events per 1K instructions in Parsnip.

Table 3 shows the frequency of the most expensive architectural events in Parsnip. With 32-entry epoch buffers, persisting epoch values to software is rare (column 2). The highest rate of epoch buffer drains is 1.38/1K instructions, in fluidanimate. Parsnip-Lines evictions are also rare (column 3). Column 4 shows that ParsnipTable rejections occur at a rate below 0.5/1K instructions in all benchmarks except fluidanimate. Despite its high rate of 79.15 ParsnipTable rejections per 1K instructions, most fluidanimate checks complete in hardware partly due to epoch buffers and the use of access capabilities (see Figure 10). Column 5 shows software checks per 1K instructions. While canneal and x264 have high software check rates of 2.57/1K instructions and 1.45/1K instructions, respectively, all other benchmarks execute fewer than 1 software check per 1K instructions.

Overall, Parsnip's access history organization and optimizations are effective in allowing most runtime checks to be performed in hardware.

5.6 Hardware Overheads

We used CACTI 5.3 [53] to model the area and latency overheads of the hardware Parsnip adds to a conventional processor core. Parsnip's storage costs are modest: the epoch buffer and Parsnip-Table for each core occupy less than 0.02 mm^2 in 32nm technology. For comparison, [54] states that a 2-wide in-order core at 32nm (excluding caches) occupies 0.1875 mm^2. Parsnip does require an extra L1D$ read port to allow data and metadata to be read in parallel, increasing access latency by about 20%. In a superscalar design, Parsnip could reuse an existing cache port but would reduce the number of memory operations that could be scheduled in parallel.

6 RELATED WORK

Hardware Approaches. The work most closely related to Parsnip is the Radish system [9] for hardware-accelerated sound and complete race detection. We compare extensively to Radish in previous sections. The LARD system [57] showed that naive usage of even sound and complete hardware data race detection can result in false and missed data races due to interactions with layers of the system stack like the OS and language runtime. LARD also demonstrates how to convey sufficient information across these layers to restore precision. Other hardware race detectors [21, 32, 36, 38, 41–43, 60] sacrifice soundness and completeness in favor of simpler and faster hardware.

The Vulcan [35] and Volition [44] hardware architectures have been proposed for detecting and recovering from sequential-consistency violations, which arise from two or more cyclically-coupled data races. These schemes provide sequential consistency at the instruction level by detecting the underlying data races that can violate SC. Both schemes implement precise data race detection, however, it is only needed during a short window within which instruction reordering can occur, which simplifies the implementation and allows for almost-negligible performance overheads. Other hardware schemes enforce stronger memory consistency models design to preserve sequential consistency or related properties [25, 30, 47, 51]. SC violation detectors ignore data races where at least one access occurs outside of the current detection window. Sound techniques like Parsnip can find these additional races, making debugging easier and supporting a wide range of race-detection clients such as record-and-replay and deterministic execution.

Beyond data races, many techniques have targeted detecting or protecting against a wider range of concurrency bugs such as atomicity violations [23, 26, 34] or general concurrency bugs [24], though these techniques suffer from occasionally reporting false bugs since the definitions of these bugs are inherently imprecise or application-specific.

Software Approaches. Many sound and complete pure-software dynamic data race detectors have been proposed [12, 40, 50]. The Parsnip design draws motivation and inspiration from this prior work, in particular FastTrack's optimized per-location metadata [15], and also [52], RedCard [16] and SlimState [56] for removal of some cross-location metadata redundancy. Despite these advances, pure-software approaches for sound and complete race detection commonly introduce slowdowns over 10x [56]. To combat performance overhead, several forms of sampling-based dynamic race detection have been proposed. Such schemes run faster but occasionally miss races [7, 11, 13, 19, 29].

In parallel to hardware proposals, software implementations of stronger serializable memory models reduce overheads relative to full software data race detection [5, 48] , but still incur notable run-time overheads and lack the benefits of full data race detection provided by Parsnip or sound and complete software systems. Some software systems have explored the use of commodity hardware transactional memory (HTM) support to assist in serializability checking [49]. Others have applied HTM to data race detection, but compared to Parsnip, they remain either unsound [59] or slow [31].

Lockset-based race detection [10, 46], an alternative to the happens-before data race detection algorithm, can report false races on code idioms like privatization or complicated locking protocols. However, the lockset algorithm can sometimes detect more races on a given execution than happens-before does. [60] proposed hardware support for lockset race detection to reduce its performance overheads.

7 CONCLUSION

With the prevalence of parallel programs and systems, efficient data race detection is an increasingly important topic. However,

previous software solutions incur high performance overheads, restricting their usability in real-world scenarios. Fast hardware race detectors either trade precision for performance, or make overly strict assumptions.

We proposed Parsnip, a sound and complete data race detector that can track memory accesses at byte granularity efficiently. Parsnip adds moderate hardware modifications to a conventional multicore processor, and does not rely on unsound type-safety assumptions. Parsnip outperforms Radish, the leading hardware race detector, by 4.6x on average. Parsnip incurs just 1.5x overhead over native execution on average.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback on prior versions of this work. This work is supported by the National Science Foundation through grant #1337174.

REFERENCES

[1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems* 28, 2 (March 2006), 207–255. https://doi.org/10.1145/1119479.1119480

[2] Sarita Adve. 2010. Data races are evil with no exceptions. *Commun. ACM* 53, 11 (Nov. 2010), 84. https://doi.org/10.1145/1839676.1839697

[3] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '04)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/1007912.1007933

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. *The PARSEC Benchmark Suite: Characterization and Architectural Implications.* Technical Report TR-811-08. Princeton University.

[5] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, Software-only Region Conflict Exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 241–259. https://doi.org/10.1145/2814270.2814292

[6] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*. Tucson, AZ, USA, 68. https://doi.org/10.1145/1375581.1375591

[7] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*. Toronto, Ontario, Canada, 255. https://doi.org/10.1145/1806596.1806626

[8] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. 2009. SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*.

[9] 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 *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 201–212. http://dl.acm.org/citation.cfm?id=2337159.2337182

[10] Anne Dinning and Edith Schonberg. 1991. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging (PADD '91)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/122759.122767

[11] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: interference-free regions for dynamic data-race detection. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12)*. ACM, New York, NY, USA, 467–484. https://doi.org/10.1145/2384616.2384650

[12] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. 245–255. https://doi.org/10.1145/1273442.1250762

[13] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=1924943.1924954

[14] Colin Fidge. 1991. Logical time in distributed computing systems. *IEEE Computer* 24, 8 (Aug. 1991), 28–33. https://doi.org/10.1109/2.84874

[15] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09*. Dublin, Ireland, 121. https://doi.org/10.1145/1542476.1542490

[16] Cormac Flanagan and Stephen N. Freund. 2013. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 255–280. https://doi.org/10.1007/978-3-642-39038-8_11

[17] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*. Tucson, AZ, USA, 293. https://doi.org/10.1145/1375581.1375618

[18] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*.

[19] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd Austin. 2011. Demand-driven Software Race Detection Using Hardware Performance Counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/2000064.2000084

[20] Jeff Huang and Arun K. Rajagopalan. 2016. Precise and Maximal Race Detection from Incomplete Traces. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[21] Ruirui Huang, E. Halberg, A. Ferraiuolo, and G.E. Suh. 2014. Low-overhead and high coverage run-time race detection through selective meta-data management. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 96–107. https://doi.org/10.1109/HPCA.2014.6835979 RaceSMM.

[22] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[23] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: detecting atomicity violations via access interleaving invariants, Vol. 41. 37–48. https://doi.org/10.1145/1168918.1168864

[24] Brandon Lucia and Luis Ceze. 2009. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. New York, NY, USA, 553–563. https://doi.org/10.1145/1669112.1669181

[25] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. 2010. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*. Saint-Malo, France, 210. https://doi.org/10.1145/1815961.1815987

[26] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. 2008. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. Beijing, China, 277–288. https://doi.org/10.1109/ISCA.2008.4

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*. New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[28] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages - POPL '05*. Long Beach, California, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[29] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09*. Dublin, Ireland, 134. https://doi.org/10.1145/1542476.1542491

[30] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFX: a simple and efficient memory model for concurrent programming languages. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*. Toronto, Ontario, Canada, 351. https://doi.org/10.1145/1806596.1806636

[31] Hassan Salehe Matar, Ismail Kuru andSerdar TaÅ§Äśran, and Roman Dementiev. 2014. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *Proceedings of the 5th Workshop on Determinism and Correctness in Parallel Programming (WODET '14)*.

[32] Milos Prvulovic. 2006. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture*.

[33] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. LogTM: Log-based transactional memory. In *Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture*. 254–265.

[34] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. 2010. AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 287–297. https://doi.org/10.1109/MICRO.2010.32

[35] Abdullah Muzahid, Shanxiang Qi, and Josep Torrellas. 2012. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 363–375. https://doi.org/10.1109/MICRO.2012.41

[36] Abdullah Muzahid, Dario SuÃąrez, Shanxiang Qi, and Josep Torrellas. 2009. SigRace: Signature-Based Data Race Detection. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/1555754.1555797

[37] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

[38] Adrian Nistor, Darko Marinov, and Josep Torrellas. 2009. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 541–552. https://doi.org/10.1145/1669112.1669180

[39] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems - ASPLOS '09*. Washington, DC, USA, 97. https://doi.org/10.1145/1508244.1508256

[40] Eli Pozniansky and Assaf Schuster. 2003. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/781498.781529

[41] Milos Prvulovic and Josep Torrellas. 2003. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th annual international symposium on Computer architecture (ISCA '03)*. ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/859618.859632

[42] S. Qi, A. A. Muzahid, W. Ahn, and J. Torrellas. 2014. Dynamically detecting and tolerating IF-Condition Data Races. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 120–131. https://doi.org/10.1109/HPCA.2014.6835923

[43] Shanxiang Qi, Norimasa Otsuki, Lois Orosa Nogueira, Abdullah Muzahid, and Josep Torrellas. 2012. Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6169039

[44] Xuehai Qian, Josep Torrellas, Benjamin Sahelices, and Depei Qian. 2013. Volition: Scalable and Precise Sequential Consistency Violation Detection. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 535–548. https://doi.org/10.1145/2451116.2451174

[45] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[46] 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 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[47] Cedomir Segulja and Tarek S. Abdelrahman. 2015. Clean: A Race Detector with Cleaner Semantics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 401–413. https://doi.org/10.1145/2749469.2750395

[48] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. 2015. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 561–575. https://doi.org/10.1145/2694344.2694379

[49] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. 2017. Legato: End-to-end Bounded Region Serializability Using Commodity Hardware Transactional Memory. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 1–13. http://dl.acm.org/citation.cfm?id=3049832.3049834

[50] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[51] Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. 2011. Efficient Processor Support for DRFx, a Memory Model With Exceptions. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11)*. New York, NY, USA, 53–66. https://doi.org/10.1145/1950365.1950375

[52] Young Wn Song and Yann-Hang Lee. 2014. Efficient Data Race Detection for C/C++ Programs Using Dynamic Granularity. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. 679–688. https://doi.org/10.1109/IPDPS.2014.76

[53] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. [n. d.]. *CACTI 5.1*. Technical Report HPL-2008-20. Hewlett-Packard Labs. http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html

[54] David Wentzlaff, Nathan Beckmann, Jason Miller, and Anant Agarwal. 2010. *Core Count vs Cache Size for Manycore Architectures in the Cloud*. Technical Report MIT-CSAIL-TR-2010-008. MIT. http://hdl.handle.net/1721.1/51733

[55] Benjamin Wester, David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. Parallelizing Data Race Detection. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[56] James Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. 2015. Array Shadow State Compression for Precise Dynamic Race Detection. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*.

[57] Benjamin P. Wood, Luis Ceze, and Dan Grossman. 2014. Low-level Detection of Language-level Data Races with LARD. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 671–686. https://doi.org/10.1145/2541940.2541955

[58] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP '05)*. ACM, New York, NY, USA, 221–234. https://doi.org/10.1145/1095810.1095832

[59] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2016. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 159–173. https://doi.org/10.1145/2872362.2872384

[60] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. 2007. HARD: Hardware-Assisted Lockset-based Race Detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. Washington, DC, USA, 121–132. https://doi.org/10.1109/HPCA.2007.346191