Efficient Architecture Support for Region-Serializability-Based Consistency

Swarnendu Biswas
University of Texas at Austin
sbiswas@ices.utexas.edu

Rui Zhang
Ohio State University
zhang.5944@osu.edu

Michael D. Bond
Ohio State University
mikebond@cse.ohio-state.edu

Brandon Lucia
Carnegie Mellon University
blucia@cmu.edu

Ohio State CSE Technical Report #OSU-CISRC-4/17-TR01

ABSTRACT

Shared-memory systems provide strong, well-defined semantics *only for* data-race-free executions. This paper proposes a new architecture design called *ARC* that provides strong, serializability-based semantics for *all* program executions, without restricting compiler or hardware optimizations. ARC employs a novel mechanism in which a core's region executes largely in isolation from other cores; ARC checks the region's consistency with the shared cache only at region end and private cache evictions. Unlike prior work that detects region conflicts, ARC does *not* serialize region commit, communicate with other cores to commit a core's region, or piggyback on M(O)ESI cache coherence.

An unoptimized design of ARC can incur high time and traffic overhead by checking consistency and ensuring coherence conservatively; we introduce optimizations that soundly eliminate much of this work and reduce costs substantially. Our evaluation shows that the resulting design compares favorably, in terms of execution time and on- and off-chip traffic, with prior work that provides the same consistency model or implements transactional memory. ARC's novel mechanism avoids the high costs and scalability bottlenecks incurred by prior work's mechanisms. ARC thus advances the state of the art in providing strong memory consistency.

1 Introduction

Shared-memory multiprocessors offer a simple, efficient execution model in which multiple software threads execute on processor cores that share a global memory. In shared-memory systems, *data races* are a fundamental barrier to the correctness of software. A data race occurs when two threads access the same shared memory location—and at least one access is a write—without using synchronization to order their accesses [6]. Data races are a problem because existing programming languages and architectures perform optimizations that *assume* data race freedom, resulting in complex or undefined semantics for executions with data races [2, 8, 23, 72, 90, 95, 101]—leading to unexpected, erroneous behaviors [2, 21, 22, 24, 25, 68].

The key to dealing with data races is providing a mechanism for detecting conflicting[1] memory accesses that is both *efficient* and *precise*. Efficient system support for a mechanism that detects or resolves conflicts—which this paper calls a *consistency mechanism*—is essential for providing comprehensible language specifications [2, 16, 30, 69, 74, 91, 97]. Efficient consistency mechanisms are also critical for enabling execution models such as transactional memory [9, 32, 50, 51, 53, 85] and system support such as concurrency bug detection [38,46] and deterministic execution [11,36,56,105]. Despite the wide-ranging value of consistency mechanisms, a precise, efficient, and practical design has eluded prior efforts (Sections 2 and 6).

This work describes a new, generally applicable, efficient consistency mechanism. We illustrate the value of our consistency mechanism by applying it to a new architecture design called *ARC*[2] that precisely detects conflicts between *unbounded* regions of a program's dynamic execution. ARC supports the *SFRSx* memory model [16, 69], which ensures that synchronization-free regions (SFRs) execute serializably or terminates the execution with a *consistency exception* that indicates the presence of a data race. It is a key contribution of this work to provide a mechanism that efficiently supports the strong SFRSx memory consistency model.

ARC's consistency mechanism isolates a region of an execution on one core from the other cores, mostly avoiding the need to send information about which data a region accessed to other cores and the shared cache. When a core's region ends, the core ensures that the region's execution saw consistent data and produced consistent results by *committing* the region's writes atomically and *validating* that the values read by the region are consistent with the values in the LLC. When a core evicts a line from a private cache to the LLC, it checks the line for consistency with the LLC and delegates further consistency checking for that line to the LLC.

When a region ends, a basic design of ARC must check the

[1]Two memory accesses to the same location *conflict* if they are executed by different threads and at least one is a write.

[2]ARC is an acronym for Architecture for Region Consistency.

consistency of all data accessed by the region and invalidate all privately cached lines to provide coherence. This basic design can incur high overhead, particularly when regions are short. We introduce optimizations that soundly eliminate most of this cost by identifying and avoiding unnecessary consistency checks and invalidations. ARC's design and optimizations are inspired in part by prior work on distributed shared memory (DSM) systems [3, 12, 18, 29, 44, 48, 61] and simplified cache coherence mechanisms [33,40,60,87, 102, 103]. The key distinction with ARC is that the prior work *assumes data race freedom*, while ARC's consistency mechanism provides guarantees for *all* executions.

ARC places few constraints on an implementation in hardware. ARC does not require a M(O)ESI cache coherence protocol, a cache coherence directory, or a direct core-to-core communication channel. Instead, ARC adds distributed *consistency controller* (CC) logic, per-byte *access information* to caches, and requires backing memory to store access information in case of LLC evictions. A banked cache of access information co-located with the LLC avoids memory lookups for an acceptable area and power overhead. ARC's CC logic is an unlikely performance bottleneck: CCs are distributed and partitioned per-core, allowing different cores' actions to proceed in parallel without core-to-core communication. Our ARC design targets the majority of commercially available CMPs with moderate core counts ($\leq$32). CMP architectures with large core counts ($>$32) are out of the scope of this work because the memory overhead to back up access information scales with core count. Multi-socket designs are also out of this paper's scope and ARC is designed for a single socket.

We evaluate ARC and compare to the consistency mechanisms from (1) *Conflict Exceptions* (CE) [69], which like ARC provides SFRSx, and (2) *Transactional Coherence and Consistency* (TCC) [50] adapted to ARC. Like ARC, TCC eschews M(O)ESI-style coherence, instead providing both consistency and coherence at region boundaries. ARC has run-time performance on par with a typical MESI system and with CE, and modest on-chip network and off-chip memory system traffic overheads that are substantially lower than CE's. In contrast with ARC, TCC's mechanism to deal with unbounded regions must frequently serialize cores' execution even at modest core counts. Crucially, ARC avoids critical communication and serialization issues faced by CE and TCC across a range of realistic core counts. Our results show that ARC advances the state of the art in architecture support for memory consistency.

2 The Need for New Consistency Mechanisms

Mainstream shared-memory systems do not detect or resolve conflicts between unbounded, concurrently executing code regions—support for which we call a *consistency mechanism*. Consistency mechanisms handle conflicts between concurrently executing regions of code, making them useful for transactional memory [9,32,50,51,53,85] and strong memory consistency models [2,16,30,69,74,91,97], as well as for system support such as error checking [38,46] and determinism [11,36,56,105]. Our work develops a new consistency mechanism; we illustrate its value by using it to support a strong, end-to-end memory consistency model, the need for which we discuss next. Section 2.2 describes several existing consistency mechanisms and their limitations, revealing the critical gap that our work fills.

2.1 Motivating Strong Memory Consistency

The specifications of shared-memory languages including C/C++ and Java are based on *DRF0*, which allows compilers and hardware to optimize code assuming no conflicts between *synchronization-free regions* (SFRs) [2, 5, 23, 72]. An SFR is a sequence of per-thread dynamic instructions delimited by synchronization operations including acquire/release, fork/join, and wait/notify operations (shown in Figure 1). As a result of this data-race-free assumption, shared-memory systems provide well-defined behaviors only for executions without data races. For such data-race-free executions, DRF0 guarantees sequential consistency (SC) [64] and *serializability of SFRs* [2, 5, 69], meaning that SFRs appear to execute atomically and in program order.

However, for executions with data races, DRF0-based language specifications provide weak or undefined semantics. In C/C++, data races have undefined semantics [2, 21, 23]. Java's memory model preserves memory and type safety in the presence of data races [72] but precludes common compiler optimizations [25, 94]. Hardware memory models such as TSO [95, 101] are generally stronger than DRF0-based language models, but they lack end-to-end guarantees.

Data races remain a real challenge in spite of much effort [1, 15, 26, 27, 34, 39, 41–43, 45, 59, 65, 73, 80, 81, 84, 99, 106, 107]. A data race's occurrence is environment, input, and timing sensitive [43, 49, 59, 99].

SFR-serializability-based consistency. Researchers have proposed support for end-to-end serializability of SFRs for *all* executions [16, 69, 83]. SFR serializability is appealing because it extends the same guarantees to executions with races that DRF0 provides only for race-free executions. Under SFR serializability, the compiler and hardware are free to optimize within SFRs. In contrast, enforcing end-to-end sequential consistency (SC) [4, 47, 66, 67, 75, 86, 96, 98, 104] or serializability of *bounded* regions [7, 30, 70, 74, 92, 97] requires restricting compiler and hardware optimizations. Architecture support alone can ensure SFR serializability, since compiler optimizations already respect SFR boundaries.

Prior work introduces consistency mechanisms to support a memory model called *SFRSx* that guarantees SFR serializability or generates a *consistency exception* [16, 69]. An SFRSx execution has the behavior of a serialization of SFRs or terminates with a consistency exception, indicating a conflict between SFRs, which *must* be because of a true data race. SFRSx makes software safer by limiting the effects of data races, and helps debug data races by making them a fail-stop condition. Figure 1 shows an execution with data races on two shared variables, x and y. An implementation of SFRSx need *not* generate a consistency exception for the read of x because the SFRs accessing x do not overlap. In contrast, the SFRs accessing y overlap, so SFRSx generates an exception because the execution may violate SFR serializability (e.g., suppose Thread 1's SFR later writes x or y).

In this work, we support SFRSx using ARC's novel architectural consistency mechanism.

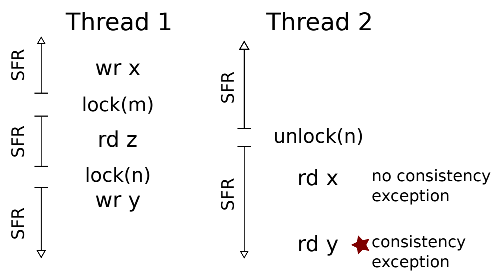


Figure 1: Under SFRSx, an execution generates an exception for a data race that may violate SFR serializability.

2.2 Existing Consistency Mechanisms

Although researchers have introduced consistency mechanisms that detect and resolve conflicts between unbounded regions, these mechanisms generally require broadcasting access information among cores or incur other serious limitations, leading to high run-time and on- and off-chip traffic costs. ARC's novel consistency mechanism addresses these limitations through a fundamentally different design.

Support for SFRSx. Existing systems provide SFRSx but with significant drawbacks. *Valor* provides SFRSx in software alone but slows executions by almost 2X on average [16]. *Conflict Exceptions* (CE) adds hardware on top of a MOESI cache coherence protocol to detect SFR conflicts, and relies on the protocol's eager messaging to exchange access metadata [69]. CE also generates prohibitively high on-chip memory traffic (Section 6.2.3) to broadcast access information to other cores at region boundaries.

Hardware transactional memory. HTM is a general mechanism for providing speculation-based, serializable execution of code regions [53,55]. HTM detects conflicts between code regions executing as *transactions*, and aborts and re-executes one or both transactions. HTM systems can use *imprecise* conflict detection, while SFRSx requires precise conflict detection; HTM must keep original copies of speculatively written data, in case of misspeculation.

Most HTMs build on M(O)ESI cache coherence protocols to detect and resolve conflicts [9,19,20,78,109,110]. The key challenge encountered by these designs, if they support unbounded transactions, is handling overflow of a transaction's working set from the private cache(s) to the shared, last-level cache (LLC), since conflicts on non-privately-cached data do not generate coherence events. Unbounded HTM designs incur substantial cost and complexity to maintain state for overflowed bits and detect conflicts as they occur. For example, *LogTM* extends the directory with "sticky" coherence states for lines in order to detect conflicts on lines that overflow a transaction [78].

Other HTMs reduce costs and complexity by supporting only *bounded* transactions, e.g., aborting transactions that overflow private caches [55, 110]. To support unbounded transactions, these HTMs require a software TM (STM) fallback [10,28,62,77].

Transactional Coherence and Consistency (TCC) is an architecture design for HTM [50] that, like our ARC design, forgoes a directory and M(O)ESI coherence, and instead provides coherence and consistency at region boundaries [50]. TCC broadcasts a transaction's write set at the end of the transaction to detect conflicts. Follow-up work to TCC shows how to alleviate this operation's inefficiency by using a directory [32] and introducing parallel commit [85]. Despite these optimizations, TCC is fundamentally limited by its use of bounded write buffers. When a transaction overflows a buffer, the executing core must execute non-speculatively, with exclusive commit access from the point of overflow to the end of the transaction [50,53]. Non-speculative, exclusive execution impedes TCC's parallelism and performance, as we show in Section 6.4.

Like TCC, *BulkSC* executes regions transactionally, using conflict detection to enforce coherence and consistency [30, 31]. BulkSC's conflict detection mechanism works by broadcasting a completing region's write set as a signature. BulkSC's regions are inherently bounded by its use of (imprecise) signatures; to ensure progress, BulkSC dynamically subdivides regions, precluding its application to a model with statically defined regions such as SFRSx or transactional memory.

3 Design Overview of ARC

This paper introduces *ARC* (Architecture for Region Consistency), an architecture that ensures serializability of executing regions using a novel consistency mechanism. In ARC's consistency mechanism, each core executes regions mostly in isolation, performing coherence and consistency actions only at region boundaries and private cache evictions. When a core's region ends, the core ensures consistency by *committing* its writes atomically and *validating* that the values read by the SFR are consistent with the values in the shared, last-level cache (LLC). The core ensures coherence by invalidating all of its privately cached lines. When a region suffers a capacity or conflict miss, the core delegates further consistency checking for that line to the LLC.

ARC applies its consistency mechanism to providing the SFRSx memory model. ARC requires minimal compiler support to identify synchronization operations, but does not restrict compiler optimizations within regions.

This section gives an overview of ARC's design and how ARC meets its goals. Sections 4 and 5 describe an architecture that implements ARC and several optimizations.

State. ARC inherently must support byte-granular tracking, which is essential for the *precise* conflict detection needed for SFRSx. Cores' private caches and the shared cache (LLC) track *access information* for each byte in a cache line that represents whether the byte has been read and/or written by a core's ongoing SFR. The LLC needs to maintain access information for lines evicted from a core's private cache[3] to the LLC. Each shared cache line maintains a *version*, which is a number incremented each time the line is written back to the LLC. Only the LLC updates a line's version, which represents the logical time of the latest write-back to the line in the LLC. Each private cache line maintains a copy of the shared line's version, for use in validating reads from the line.

Actions at reads and writes. When a core reads or writes a byte of memory, it updates the access information (read or

[3]For simplicity of exposition, this section abstracts a core's L1 and L2 caches as a single private cache. In Section 4's architecture design, each core has L1 and L2 private caches.

write bit) for the accessed byte in its private cache. Note that a region's read or write does *not* trigger any communication with the LLC or other cores, regardless of whether the LLC or other cores' caches have valid copies of the line.

If a core evicts a line that has access information, the core's private cache writes back the access information to the LLC, along with the line data if the line is dirty. The LLC uses the access information to detect conflicts with other cores that have evicted the same line. It saves the access information at the LLC to detect conflicts with other cores when they validate reads, commit writes, or evict lines to the LLC. Note that when a core evicts a private line, the core and LLC do *not* communicate with other cores.

Actions at SFR boundaries. When a core's region ends, it ensures serializability by validating its reads and committing its writes atomically, using a *region commit protocol* that consists of the following three operations in order:

(1) Pre-commit: The core sends its *write* access information to the LLC, which checks for conflicting access bits for the same byte in the LLC, which indicate a conflict (in which case ARC generates a consistency exception). The LLC maintains the core's write access information during the next step, read validation, to ensure the atomicity of validating reads together with committing writes.

(2) Read validation: The core needs to validate that the values it read are consistent with the LLC's current values. Sending data values would generate a lot of traffic—and comparing values alone would be unsound, by not validating with respect to a single LLC snapshot. ARC avoids these pitfalls with a combination of version and value validation.

Algorithm 1 shows how read validation works. For each private cache line to be validated, the core sends the line's version to the LLC for comparison. For simplicity, the algorithm depicts a synchronous reply for each validation request; in fact, the LLC can reply asynchronously, and it need not reply at all if the versions match (Section 4).

A version mismatch indicates a write to the same line but not necessarily the same bytes that the validating core has read. The core handles a version mismatch by checking that the *data values* that it read match the current values in the LLC (using the LLC's data values, sent in the LLC's response). The core also ensures the absence of a write–read conflict by checking that no locally read byte has its write bit set in the LLC (using the LLC's write bits, sent in the LLC's response). If either check fails, the core generates a consistency exception. The core updates the line's version to the new value, to avoid the same mismatch when validation retries. A core may repeatedly retry read validation and experience starvation, but ARC is *livelock and deadlock free*: a version mismatch means that some other core made progress by writing to the LLC.

(3) Post-commit: The core writes back dirty bytes to the LLC and clears its private access information. The LLC clears all of its access information for the core. In addition, the core must *invalidate all lines* in its private cache, to ensure coherence at SFR boundaries. This *self-invalidation* step degrades locality, especially for short regions; Section 5 introduces optimizations that reduce self-invalidation costs.

Algorithm 1 A core performs read validation

```
 1: repeat
 2:   mustRevalidate ← false
 3:   for all private cache lines L with a read-only byte do
 4:     let v ← getVersion(L)
 5:     Send L's address and v to LLC
 6:     resp ← LLC's response                ▷ resp is ⊥ or ⟨v′, w′, d′⟩
 7:     if resp ≠ ⊥ then
 8:       ⟨v′, w′, d′⟩ ← resp  ▷ LLC line's version, write bits, & data values
 9:       mustRevalidate ← true
10:       d ← getData(L)
11:       if d′ ≠ d ∨                         ▷ Compares read-only bytes only
            w′ ∩ getReadBits(L) ≠ ∅ then
12:         Consistency exception!
13:       end if
14:       setVersion(L, v′)
15:     end if
16:   end for
17: until not mustRevalidate
```

We note that, unlike some other consistency mechanisms such as TCC's [50] (Section 2.2), the region commit protocol for a core can proceed *in parallel* with other cores performing the region commit protocol or executing regions. To see why the region commit protocol need not be serialized, consider the following two insights. First, the core and the LLC do *not* communicate with other cores' caches during the region commit protocol. Second, the region commit protocol ensures atomicity of a core's writes and reads by setting a core's write bits in the LLC for the duration of the region commit protocol.

Write-after-read upgrades. ARC's use of value validation requires careful handling of *write-after-read (WAR) upgrades*. A WAR upgrade happens when a core writes a byte that it read earlier in its ongoing region. Simply overwriting the byte in the private cache line would make it impossible to value-validate any read performed earlier in the region. ARC thus sends an upgraded line's read access information and version to the LLC, before the write happens. The LLC immediately read-validates the line and detects future read–write conflicts for the line (similar to how private cache line evictions are handled). As Section 4 describes, the ARC architecture avoids communicating with the LLC for WAR-upgraded L1 lines, by preserving an unmodified copy of the line in the L2.

4 Architecture Design of ARC

The ARC architecture is a collection of modifications to a multi-core processor. In the *base* architecture (i.e., without ARC's modifications), each core has a cache hierarchy with private, write-back L1 and L2 caches; cores share the last-level cache (LLC); and there is *no support for cache coherence*. Each cache line has only a *valid* bit and a *dirty* bit. The private caches are inclusive and the LLC is not inclusive. Figure 2 shows the components that ARC adds to the processor: (1) access information storage and management and (2) a distributed consistency controller (CC).

4.1 Private Access Information Management

Each core maintains access information for each of the lines in its private caches, as Figure 3 shows. Each L1 and L2 line has a 32-bit version that ARC uses to detect consistency violations. ARC associates two bits per byte with each line

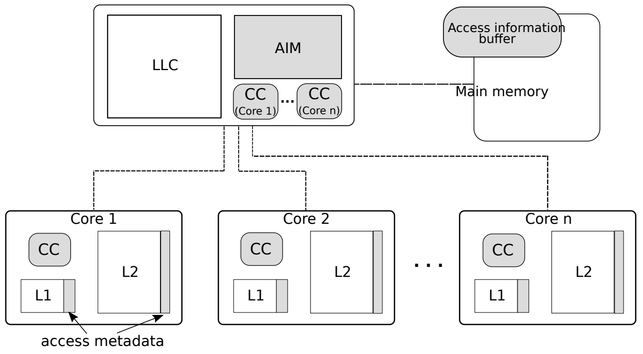


Figure 2: The ARC architecture (not according to scale). The parts shaded gray are hardware structures added by ARC. The consistency controller (CC) is distributed across the architecture.

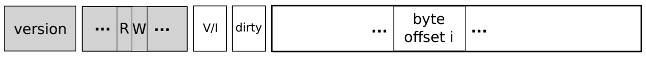


Figure 3: Per-line metadata introduced by ARC for private caches. Metadata added by ARC is shaded gray.

in the core's L1 and L2 caches. A byte's *read bit* indicates that the byte was read, without first being written, during the current SFR. A byte's *write bit* indicates that the byte was written during the SFR.

Updating access information. When a core writes a byte, it sets the byte's write bit if it is not already set. When a core reads a byte, it sets the byte's read bit only if the byte's read and write bits are unset.

A *write-after-read (WAR) upgrade* occurs when a core writes an L1 byte that was previously read in the same region. ARC must *preserve* a WAR-upgraded byte's original value for use during value validation. ARC relies on having the original value in the L2; it copies the line's access information to the L2 before the write executes. When a core evicts a dirty L1 line to the L2, ARC checks the L2 line's access information. If the L2 line has a read-from byte that was written in the evicted L1 line, ARC immediately validates reads to the line using the mechanism described in Section 4.3.2.

Evictions. When a core evicts a line from the L1 to the L2 cache, the line's access information is copied to an identical bit array for the line in the L2. When the L2 evicts a line, it sends the line's access information to the *access information memory* (AIM), co-located with the LLC; Section 4.2 describes the operation of the AIM and the LLC.

4.2 LLC Access Information Management

Rather than storing access information for each LLC line in the LLC or in memory, ARC stores the information in a cache-like memory called the *access information memory* (AIM). For each byte of access information in a cache line, an AIM entry contains one read bit for each core (C bits, where C is the number of cores in the processor), and the current writer core if any (1 bit to indicate whether there is a writer core and $\lg C$ bits for the writer core). An AIM entry also contains a 32-bit *version*, which is used during read validation. For a

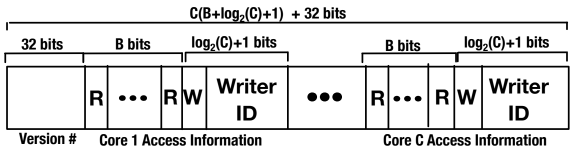


Figure 4: An AIM entry for a processor with C cores and B-byte cache lines.

processor with C cores and B-byte cache lines, a cache line's AIM entry is $32+(C+1+\lg C)\times B$ bits: 100 bytes per entry for an 8-core processor with 64-byte cache lines. Figure 4 illustrates the structure of an AIM entry.

When a core writes back a line to the LLC, the LLC updates the line's AIM entry to reflect the line's access information. The LLC copies the core's updated access information from the private line's metadata into the core's access information bits in the line's AIM entry. When a core writes back a dirty line to the LLC, the LLC also increments the version for the line that is stored in the AIM. Note that only the LLC, not any of the private caches, updates a line's version.

AIM design. As a centralized structure, contention by cores to access the AIM threatens scalability at very high core counts. Banking the AIM is straightforward, reduces contention, and mitigates the threat to scalability. Our design assumes an AIM with 8 banks.

An "ideal" AIM would contain one entry for every line in the LLC. However, an ideal AIM is impractical: in a system with 8 cores, 64-byte lines, and a 16MB, 16-way LLC, the AIM would be around 25MB—an impractically large (103 mm^2), slow (7.4ns), and power hungry (0.9W leakage per bank) on-chip structure in 32-nm technology (data from CACTI 5.3 [63]).

Instead, ARC uses a realistic AIM design that, for 8 cores, has 32K entries and 4-way associativity. This 3.2MB AIM has implementable area (23 mm^2), latency (2.4ns), and leakage power (140mW per bank). In a high-end, 32-nm Intel Core i7-3970X at 3.5GHz [57], the AIM adds about $\sim$1% overhead to the total 2,362 mm^2 package area, 9-cycle access latency (easily hidden by LLC latency), and leakage at a tolerable 0.75% of TDP.

The size of an entry in the AIM scales with the number of cores, and the AIM's size, latency, and leakage power limit ARC's scalability. The AIM is unlikely to scale to CMPs with very large numbers of cores (i.e., $>$32) and our design targets the majority of commercially available CMPs that have moderate core counts (i.e., $\leq$32 cores). The AIM's hardware design scales well across this range of moderate CMP core counts. At 16 cores, a 32K-entry AIM is realizable with a 3.5ns access time, 56 mm^2 area overhead, and 0.3W leakage power per bank (data from CACTI 5.3 [63]). At 32 cores, a 32K-entry AIM is costly, but realizable with a 5.9ns access time, 163 mm^2 area cost, and 0.80W leakage power per bank. A less costly 16K-entry AIM for a 32-core machine has a latency of 5.6ns, area of 149 mm^2, and leakage power of 0.65W per bank. The AIM remains effective across core counts: for 32 cores, the smaller, 16K-entry AIM design performs competitively with the ideal AIM (Section 6.2.3).

Virtualizing access information to memory. Regardless of the geometry of the AIM (ideal vs. cache-like), ARC must preserve the access information for AIM entries evicted from the AIM. Similar to prior work [69], ARC maps evicted AIM entries into a dedicated region in memory.

To enable post-commit to clear all access information for a core without explicitly tracking and updating access information that has been evicted to memory, ARC augments an evicted AIM entry with a list of saved *epochs*, one per core, before pushing the entry to memory. An epoch is a number that identifies a core's SFR. The AIM maintains an epoch for each core in a *current epoch register*. A core's current epoch register in the AIM is incremented whenever a core finishes an SFR. When the AIM fills a line, it compares the incoming entry's saved epochs to each core's current epoch register. If the epochs differ, the AIM clears the access information for that line for that core. It is then correct to clear a core's access information because the saved epoch indicates that the access information represents accesses from a previous SFR.

For a system with C cores, B-byte cache lines, 32-bit versions, and E-bit epochs, ARC must preserve $P = (32 + (C + 1 + \lg C) \times B + E \times C)/B$ bits per byte of access information from the AIM. ARC reserves the high-order $\arg\min_i (2^i - 1 \geq P/8)$ address bits and uses addresses with those bits set to store evicted access information. With 8 cores, 64-byte lines, and 32-bit epochs, a system needs $P = 16.5$ bits per byte of backing memory. In such a system, ARC reserves memory addressed by the most significant 2 address bits for access information, leaving the application with a 62-bit address space. Although our prototype implementation reserves physical memory for an AIM entry for every line, an implementation could use virtual addresses for backing memory or use a sparse representation for AIM entries in memory (like LogTM [78]), potentially at higher cost.

4.3 Consistency Controller (CC)

ARC ensures consistency using a *region commit protocol* that is implemented in ARC's distributed *consistency controller* (CC). Section 3 described the basic steps of the region commit protocol. Here we focus on the CC's implementation.

The CC consists of per-core logic, which is unlikely to limit scalability. The CC has a read/write interface to the AIM that allows the CC and AIM to exchange access bits, versions, and values. The CC also includes *AIM-side* buffering and comparison logic that is co-located with the AIM. AIM-side logic is replicated per-core to avoid scalability issues.

4.3.1 Region Commit Protocol

A core's CC initiates the region commit protocol. A core's protocol phases can overlap with other cores performing the protocol or executing regions; the protocol ensures atomicity by setting a core's write bits in the AIM during pre-commit and not clearing them until post-commit. During the protocol, a core's CC need *not* ever communicate directly with other cores. A core's CC checks consistency using only (1) data in the LLC, (2) metadata in the core's cache, and (3) metadata in the AIM.

Pre-commit. During pre-commit, the core streams access information from its dirty, privately cached lines to its AIM-side CC logic. The AIM-side CC logic buffers the lines' write access information while it reads in the line's access information from the AIM. The core's AIM-side CC logic compares the core's access bits to all other cores' access bits using fixed-function logic. The logic selects the core's access bits and computes a bitwise `and` of those bits with all other cores' bits. If a logical `or` of the bits in the result is nonzero, then the bits indicate a conflict and the core's CC delivers an exception to the core. If not, the AIM-side CC logic updates the access bits in the AIM to match the buffered ones it received from the core's CC.

Read validation. After pre-commit, the CC begins read validation. The core's CC streams a sequence of messages to its AIM-side CC logic, one for each line the core read during the ending region. Each message contains the line's address and version from the core's private cache(s). The core's AIM-side CC logic fetches addresses and versions from the AIM for each message it receives from the core, and uses dedicated logic to compare the line's version in the core's message to the version from the AIM. If all versions match and no write bits are set for a remote core for any offset in the shared line, read validation completes successfully. If a read line's versions match, but a write bit was set by a remote core, the core's AIM-side CC logic responds to the core's CC with read bits and checks for write–read conflicts. In case of a conflict, the core raises a consistency exception.

If a line's version differs, then another core wrote the line during the ending region and there may be a conflict. On a version mismatch, the core's AIM-side CC logic sends the core's CC the line's address and updated version. The core re-fetches that line from the LLC into a dedicated *line comparison buffer* in the core. The core compares the (read-only) line in the private cache to the line in the comparison buffer.

If the lines differ, then the validating core read inconsistent data and raises a consistency exception. If they match, then the core may have seen consistent data in its region. On a version mismatch, the core also sets its *revalidate bit*. The revalidate bit indicates that after the core finishes validating all remaining lines, it must start again from the beginning, streaming version messages to its AIM-side CC logic, to ensure that it saw consistent data. After the core completes validation without version mismatches, it unsets the revalidate bit and continues.

Post-commit. During post-commit, a core prepares for executing its next region. The core streams dirty bytes in its L1 and L2 caches to the LLC, and it clears all access information and invalidates all lines in its L1 and L2 (e.g., using gang clearing [76]). The core's AIM-side CC logic clears the core's access information in the AIM. For lines evicted to memory, the AIM clears access information *lazily* when the line is next cached in the LLC. Finally, the AIM increments the committing core's epoch.

4.3.2 Other CC Responsibilities

Handling evictions to the LLC. When an L2 evicts a line with access information, the core's CC performs pre-commit and read validation on the line. The CC checks for conflicts using the access information in the AIM, and checks that the L2 line's bytes match the version or (if not the version) the values in the LLC. Finally, the AIM-side CC logic updates the line's access information in the AIM.

When a core's L2 fetches an LLC line with access bits in the AIM for that core, the LLC sends the core the line's data values and the access bits for the core, which the core uses to populate its L1 and L2 access information.

Delivering consistency exceptions. When a core's CC detects or infers a conflict, ARC generates a consistency exception, by raising a dedicated per-core signal (via a non-maskable interrupt) for the core that detected the conflict. By default, the core that receives the interrupt executes operating system code to terminate program execution.

4.4 Implementing Synchronization

By forgoing M(O)ESI coherence, ARC needs a mechanism to implement lock operations such as acquire and release. ARC's mechanism for implementing locks follows DeNovoND's mechanism for locks, which uses distributed queue-based locks to solve the same problem [103]. The main difference is that DeNovoND's lock protocol allows core-to-core communication, while ARC uses only communication between the core and LLC.

We assume compiler support to identify lock operations as region boundary operations that should be handled specially (similar to endR in CE [69]). Alternatively, ARC could handle legacy programs by using a pthread library implementation modified to identify lock operations for ARC, but this approach would not support non-pthread synchronization (e.g., inline assembly with atomic accesses).

5 Design Optimizations

At the end of the region commit protocol, a core's CC invalidates all lines in its L1 and L2. This *self-invalidation* step is a key source of overhead in the ARC design. Section 5.1 introduces optimizations that focus on reducing self-invalidation soundly. Section 5.2 describes how to reduce traffic generated by the region commit protocol.

5.1 Avoiding Self-Invalidation

We introduce separate optimizations for *touched* lines (read or written by the committing region) and *untouched* lines.

Touched lines. The intuition for avoiding invalidation of touched lines is that pre-commit and read validation already process these lines and can check if they are up-to-date and thus do not need to be invalidated. ARC need not invalidate privately cached lines that are *read-only*. This optimization is correct because read validation already ensures that read-only lines in the private cache are consistent with the LLC's copy.

For *dirty* lines, pre-commit checks if a line's version is unchanged in the LLC—a sufficient condition for not invalidating the line. This optimization extends pre-commit to send the core's cached version of the line to the core's AIM-side CC. If the version matches the value in the AIM, then the core has the latest version and does not need to invalidate the line. On a version mismatch, the AIM-side CC sends a message asynchronously to the core indicating that it must in fact invalidate the line.

Untouched lines. An untouched line need not be invalidated if ARC can ensure that other cores have not written to the line during the region's execution. We introduce two optimizations that exploit this insight.

The first optimization adds *cond-invalid* as a new state for private cache lines in addition to *valid* and *invalid*; *cond-invalid* indicates that the line's data is valid only if the LLC's version is unchanged. During post-commit, a core changes each untouched line's state to *cond-invalid*, instead of *invalid*. When a core accesses a line in the *cond-invalid* state for the first time in a subsequent region, the core's CC sends its copy of the line's version to the core's AIM-side CC, which compares the L2's version with the AIM's version for the same line and replies to the core indicating whether the versions match. If the versions match, the core's CC upgrades the line in the L2 and L1 caches to *valid*. Otherwise, the access is handled as a miss. This optimization reduces on-chip traffic by often sending only a version rather than data values on an L2 cache miss.

Second, ARC minimizes self-invalidations for untouched lines by keeping a per-core *write signature* [31] for the core's AIM-side CC that encodes *which* lines have been updated in the LLC during the core's current region *by any other core*. During post-commit, if a line is not in a core's write signature, the core's CC need not invalidate the line.

The AIM-side CC encodes a write signature for each core's ongoing region as a Bloom filter [17,31]. Whenever a core writes back to the LLC, the AIM-side CC updates *every other* core's write signature to include the updated line. When a core starts read validation, the AIM-side CC sends the core's CC its write signature and clears the AIM-side CC's copy of the core's signature. The core's CC uses its received copy of the write signature during post-commit to identify untouched lines in its private caches. If the signature does not contain the line, then it was definitely not updated in the LLC during the core's execution, so it can stay in the *valid* state in the core's private caches.

A small Bloom filter is desirable because the AIM-side CC sends it to the core's CC at each region commit. A small Bloom filter is sufficient to encode a write signature for short regions—which benefit the most from reducing self-invalidation. The AIM-side CC uses a 112-bit Bloom filter for each core, which along with control data fits into one 16-byte network flit (Section 6.1 and Table 1). We use two hash functions that each set one Bloom filter bit.

5.2 Optimizing the Region Commit Protocol

The following optimizations minimize the work performed by the region commit protocol.

Optimizing read validation. A core c can forgo validating a line if the line was not updated in the LLC by any other core during c's region. To do this check, c's CC uses the per-core write signature introduced in Section 5.1, obtained before read validation starts. To ensure atomicity, c's CC re-fetches the write signature *after* read validation to ensure it has not changed; if it has, c's CC restarts read validation.

Deferring write-backs. We optimize post-commit's write-back of dirty lines, by deferring sending the data to the AIM-side CC until (and if) another core needs it. ARC implements this optimization by adding $\lg C$ additional bits (for a system with C cores) to each cache line in the LLC to identify the *last-writer* core that has up-to-date data, plus an additional bit to indicate whether the line's state is *deferred*. If another core requests a deferred line from the LLC, the LLC first fetches the latest values from the last-writer core. This optimization, which is analogous to the Owner state in the MOESI protocol [100], provides substantial benefit by avoiding obligatory write-backs at every region commit.

6 Evaluation

This section evaluates performance and on- and off-chip traffic for ARC, compared with competing approaches.

6.1 Simulation Methodology

We implemented ARC in a simulator based on the RADISH simulator provided by its authors [37]. For comparison to a baseline, we have implemented a directory-based MESI cache coherence protocol [100] to model current shared-memory systems, which we call *MESI*. We have also implemented *Conflict Exceptions* (CE) [69] on top of MESI. The simulators consume a serialized trace of events generated by a Pintool [71]. Multiple simulator configurations process the *same* trace, in order to eliminate differences due to run-to-run nondeterminism. All three simulators model a realistic baseline architecture with 8–32 cores, detailed in Table 1.

For ARC, the LLC is *not* inclusive (Section 4), and the simulator models a realistic AIM cache, as Table 1 shows. For MESI and CE, the LLC is inclusive in order to support a directory protocol [100], and the directory is embedded in the LLC with the same associativity as the LLC (see Figure 8.6 in [100]). The CE algorithm requires memory access on private cache evictions to back up access metadata and to fetch access metadata on LLC hits under certain conditions [69]; our CE simulator optimistically assumes that the latency of accessing memory is masked by subsequent operations. The simulators treat each pthread function call as a corresponding lock operation. The ARC simulator treats other atomic accesses (i.e., instructions with the x86 LOCK prefix) as special accesses that are ignored by ARC's consistency mechanism and handled like locks (Section 4.4), except that the accesses do not delineate regions.

Section 6.4 compares the mechanisms of ARC and TCC [50] by estimating the costs of using TCC's mechanism, instead of ARC's mechanism, to provide SFRSx.

Estimating execution time. Table 1 shows the number of cycles required for memory and non-memory instructions. All three simulators report the maximum number of cycles for any core; as in prior work [14, 37], cores do not model time spent waiting at synchronization. All three simulators model wait-free, write-back caches with idealized write buffers.

The ARC simulator models the costs of ARC performing operations at region boundaries. Since cores send multiple messages without waiting synchronously for responses during the pre-commit and read validation phases, we compute the cycle cost of messaging based on the total size of messages sent and the available bandwidth between a core and the LLC. During read validation, a core sends lines' versions to the LLC. Each 16-byte flit contains four lines to be validated, since a flit can fit four tags and versions plus a control block.

Processor	8-, 16-, or 32-core chip at 1.6 GHz. Each non-memory-access instruction takes 1 cycle.
L1 cache	8-way 32 KB per-core private cache, 64 B line size, 1-cycle hit latency
L2 cache	8-way 256 KB per-core private cache, 64 B line size, 10-cycle hit latency
Remote core cache access	15-cycle one-way cost
LLC	64 B line size, 35-cycle hit latency
8 cores:	16-way 16 MB shared cache
16 cores:	16-way 32 MB shared cache
32 cores:	32-way 64 MB shared cache
AIM cache	4-way metadata cache with 32K lines and 8 banks
8 cores:	100 B line size (∼3.2 MB), 4-cycle hit latency
16 cores:	172 B line size (∼5.4 MB), 6-cycle hit latency
32 cores:	308 B line size (∼9.7 MB), 10-cycle hit latency
Memory	120-cycle latency
Bandwidth	NoC: 100 GB/s, 16-byte flits; Memory: 48 GB/s

Table 1: Architectural parameters used for simulation.

		Avg. accesses per SFR ($\times 10^3$)		
	Threads	$n = 8$	$n = 16$	$n = 32$
blackscholes	$1+n$	8,560	4,280	2,140
bodytrack	$2+n$	49.9	39.7	29.9
canneal	$1+n$	498	130	64.9
dedup	$3+3n$	44.7	44.6	52.5
ferret	$3+4n$	995	832	370
fluidanimate	$1+n$	0.115	0.085	0.055
raytrace	$1+n$	4,740	2,470	1,260
streamcluster	$1+2n$	1.37	0.428	0.122
swaptions	$1+n$	78,000	39,000	19,500
vips	$3+n$	88.6	68.6	50.7
x264	$1+2f$	0.487	0.486	0.560

Table 2: Threads spawned and average region sizes in thousands (rounded to 3 significant figures unless < 0.1) for the PARSEC benchmarks. n is the minimum threads parameter in PARSEC. f is the input-size-dependent number of frames processed by x264.

We assume that the core's AIM-side CC and LLC are ported to handle a flit's four validation requests at a time. The ARC simulator models version mismatches, including the costs of the AIM-side CC alerting the core's CC and the core's CC restarting and repeating read validation. The ARC simulator models post-commit assuming gang-clearing for self-invalidation of private cache lines and bulk-clears of per-core AIM information.

Estimating network traffic. We simulate an on-chip network and off-chip memory network with 16-byte flits and the bandwidth characteristics shown in Table 1. Control messages are 8 bytes (tag plus message type); a MESI data message is 64 bytes (corresponding to a cache line). For ARC write-backs, we model idealized write-buffer coalescing that sends only the dirty bytes in a line.

Benchmarks. Our experiments execute the PARSEC benchmarks [14], version 3.0-beta-20150206, with simmedium inputs. We include 11 of 13 programs; freqmine uses OpenMP as the parallelization model, and our Pintool fails to finish executing facesim. We report cycles and traffic for the parallel "region of interest" (ROI) only [13]; vips lacks an ROI annotation so we use its entire execution as the ROI.

Table 2 shows how many threads each benchmark spawns, parameterized by *n*, which is PARSEC's *minimum threads* parameter (the -n flag). The last three columns show the average number of memory accesses performed per SFR for *n*=8, *n*=16, and *n*=32. The simulators set *n* equal to the number of cores (8, 16, or 32) in the simulated architecture. The simulators map threads to cores using modulo arithmetic.

Consistency exceptions. When the ARC simulator detects the conditions for a consistency exception, it logs the exception and continues execution normally. The simulator can report both locations involved in a region conflict, since it maintains the last-access source location corresponding to each read and write bit of access information. The following table shows, for 8 cores, the number of distinct conflicts (i.e., unique unordered pairs of static source locations) and dynamic conflicts for the two benchmarks for which ARC generated consistency exceptions.[4]

	Distinct conflicts	Dynamic conflicts
canneal	1	1,205
streamcluster	11	836

Using Google's ThreadSanitizer [93] and by implementing "collision analysis" [42] in Pin, we have confirmed that each detected conflict corresponds to a true data race.

6.2 Run-Time Performance and Traffic

Figures 5 and 6 show our main results. The *MESI-8*, *CE-8*, and *ARC-8* configurations show the performance and traffic overheads running on 8 cores. Correspondingly, the next two groups of three bars show results on 16 and 32 cores. Each bar is the mean of three trials, and the results are normalized to MESI with 8 cores.

6.2.1 Run-Time Performance

Figure 5(a) shows executed cycles as reported by the simulators, broken down into different components. *MESI* and *CE* (which builds on MESI) are divided into cycles attributed to *coherence* and *other execution*. Coherence cycles are those spent when the directory forwards requests to remote cores and for core-to-core communication. For ARC, cycles are divided into cycles for *pre-commit*, *read validation*, and *post-commit*, and cycles for *region execution*.

Figure 5(a) shows that CE adds negligible performance overhead over MESI. Our CE simulator ascribes no additional cost for transmitting access metadata piggybacked on MESI coherence messages, nor to access metadata in memory; hence the performance of CE is similar to MESI.

The figure shows that ARC outperforms MESI and CE in several cases by avoiding the latency of MESI coherence. ARC underperforms MESI and CE in a few cases (fluidanimate, and streamcluster with 16 and 32 cores) since regions are short (see Table 2) and incur latency from cache misses due to frequent self-invalidation. On average ARC outperforms MESI and CE by 4–5% for 8 cores and performs nearly identically for 16–32 cores.

[4]We ignore conflicts on internal library and system data.

	On-chip			LLC-to-memory		
	MESI	CE	ARC	MESI	CE	ARC
bodytrack	1	2	3	< 1	< 1	< 1
canneal	60	62	54	2	17	5
dedup	3	3	3	1	1	1
ferret	5	5	4	< 1	2	< 1
fluidanimate	7	178	35	< 1	1	< 1
streamcluster	13	90	18	< 1	10	< 1
vips	9	9	8	1	4	2

Table 3: Average on- and off-chip bandwidth, in GB/s, for MESI, CE, and ARC for 32 cores. We omit benchmarks for which every value in the row would be $\leq$2 GB/s.

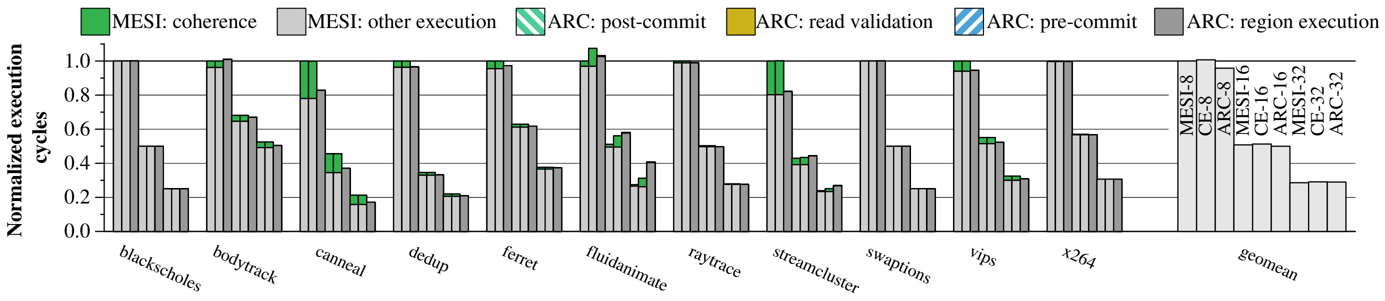
6.2.2 On-Chip Traffic

Figure 5(b) compares the on-chip network traffic incurred by MESI, CE, and ARC, counted in 16-byte flits. On-chip traffic for ARC is defined as all communication between cores and the LLC/AIM. For MESI and CE, it is communication between cores and the LLC, and core-to-core communication.

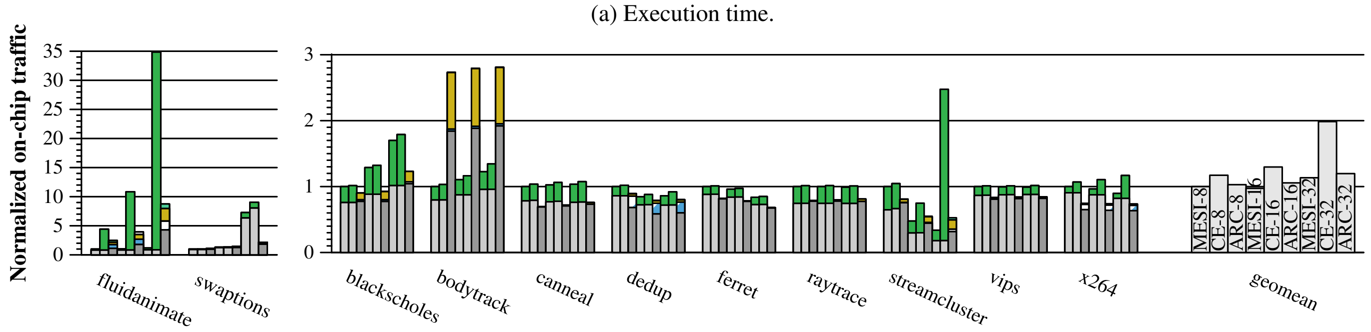
The key result from Figure 5(b) is that for all benchmarks except fluidanimate, ARC's traffic overhead increases proportionately with core count, and ARC's traffic scalability is nearly identical to MESI's. This result shows that ARC's traffic overhead is unlikely to prevent scaling to moderate core counts. For fluidanimate, traffic increases disproportionately with core count for both CE and ARC because fluidanimate performs more writes and synchronization operations with increasing numbers of threads. With 16 worker threads, fluidanimate has 14% more writes and 44% more synchronization operations compared to 8 worker threads, while it has 46% more writes and 147% more synchronization operations with 32 threads. Thus, the benchmark has progressively smaller regions with more threads (Table 2). As described in prior work [69] and emulated in our simulator, the CE protocol piggybacks on MESI coherence messages and transfers access metadata (endR messages [69]) at region boundaries to detect region conflicts. For CE, more frequent region boundaries imply more frequent exchange of access information among cores [69], which results in CE incurring more on-chip network traffic than MESI, especially for fluidanimate and streamcluster. For ARC, more frequent region boundaries cause more frequent invocations of pre- and post-commit, read validation, and self-invalidation operations, which all increase traffic, as the figure's component breakdown shows.

For swaptions with 32 cores, MESI and CE incur substantially more traffic than ARC. This result is due to swaptions's frequent LLC evictions, for which an inclusive LLC (required by MESI and CE but not ARC; Section 6.1) must recall privately cached copies of the line.

Is the *raw magnitude* of traffic used by MESI, CE, or ARC a cause for concern? The *On-chip* columns in Table 3 show the average on-chip bandwidth used, in GB/s, for MESI, CE, and ARC on 32 cores. For fluidanimate and streamcluster, the CE algorithm incurs *high* on-chip network traffic (178 and 90 GB/s), saturating or nearly saturating the on-chip network's available bandwidth (100 GB/s). ARC uses several times less bandwidth than CE for fluidanimate and streamcluster, and its usage for every program is significantly less than the available bandwidth.



(a) Execution time.



(b) On-chip network traffic.

Figure 5: Execution time and on-chip traffic costs for MESI, CE, and ARC for 8–32 cores, normalized to MESI with 8 cores. The suffix for each simulator indicates the number of cores. The legend at top applies to both graphs.

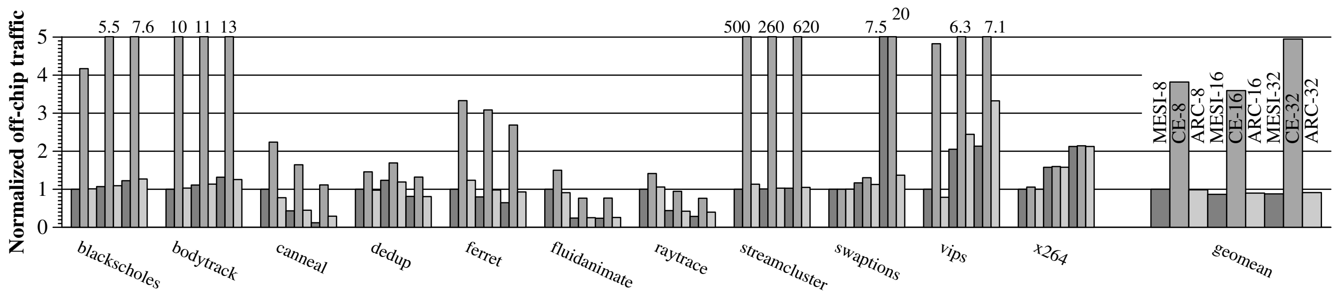


Figure 6: LLC-to-memory traffic for for MESI, CE, and ARC for 8–32 cores, using the same configurations as Figure 5. The different shades of gray differentiate MESI, CE, and ARC, and are *unrelated* to Figure 5's legend.

6.2.3 *Off-Chip (LLC-to-Memory) Traffic*

Figure 6 shows the LLC-to-memory (off-chip) traffic for MESI, CE, and ARC. CE incurs high off-chip traffic overhead over MESI, since the design backs up and fetches evicted access metadata information to and from memory [69]. In particular, CE must back up access bits in an *in-memory* table when a line that was accessed in an ongoing region is evicted from a private cache or the LLC. Similarly, the CE design requires memory traffic even on an LLC hit, if the line was evicted to memory or the core has evicted a line from its private caches during the ongoing region. Table 3's *LLC-to-memory* columns show that CE requires high network bandwidth for canneal and streamcluster compared to MESI and ARC. The memory traffic requirement for ARC is comparable to MESI for all benchmarks.

Sensitivity to AIM cache size. Our experiments by default use an AIM cache with 32K entries (Table 1). We have also evaluated ARC with an *idealized* AIM cache that has one entry for each LLC line, as well as a 16K-entry AIM cache (results not shown). On 32 cores, the idealized AIM cache reduces memory traffic by 5.8% on average, compared with the 32K-entry AIM cache, confirming that a 32K-entry design is reasonably close to the ideal case. With a lower hardware cost, a 16K-entry AIM cache increases memory traffic by only 5.7% on average, compared to the 32K-entry design. The impact on execution time is minimal: on average, the idealized AIM improves performance by less than 1%, and the 16K-entry AIM degrades performance by less than 1%.

6.3 Impact of Optimizations

The default ARC configuration includes all optimizations presented in Section 5. This section compares with ARC *without* optimizations, focusing on on-chip network traffic since that metric is most affected by the optimizations. Figure 7 shows the on-chip network traffic incurred by MESI and different configurations of ARC, for 8 cores, normalized to MESI. *ARC unopt* includes none of Section 5's optimizations; it incurs 61% more traffic than MESI on average. For fluidanimate, *ARC unopt* incurs high on-chip traffic relative to MESI. *ARC inv opt* uses only the optimizations for reducing self-invalidations (Section 5.1), thereby reducing traffic sub-

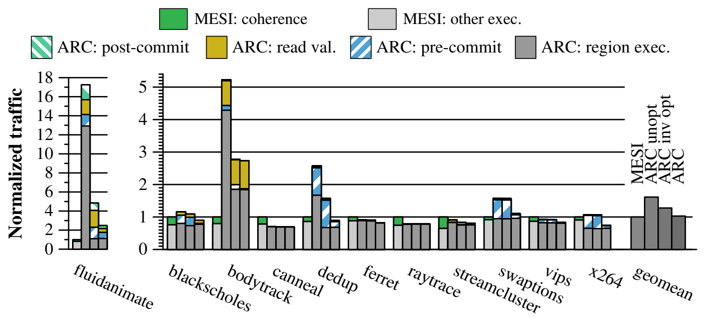


Figure 7: The effect of ARC optimizations on on-chip network traffic in a system with 8 cores.

stantially for fluidanimate and other programs. On average, *ARC inv opt* incurs 28% more traffic than MESI. The last configuration, (fully optimized) *ARC*, reduces traffic further by optimizing commit and read validation, incurring only 2.9% average traffic over MESI for 8 cores.

6.4 Comparison with TCC

Like ARC, TCC provides both consistency and coherence at region boundaries [50] (Section 2.2). However, ARC's novel consistency mechanism addresses essential flaws in TCC related to buffer bounding limitations, eliminating a key impediment to parallel performance. Furthermore, ARC avoids the costs of detecting conflicts directly, instead inferring conflicts by validating reads of privately cached data.

To compare TCC with ARC empirically, we evaluate a modified version of ARC called *ARC-TCC* that uses TCC's mechanisms and algorithms. For ARC-TCC, we compute execution cycles and on-chip traffic *excluding* pre-commit, read validation, and post-commit, but *including* the following: each region broadcasts its write set, and a region that overflows its private caches cannot execute in parallel with other overflowed or committing regions [50] (Section 2.2). We model other costs (e.g., private and shared cache hits and misses) in the same way as for ARC.

For 8 cores, we find that ARC-TCC increases execution cycles by 2.8X and on-chip traffic by 3.8X on average for all programs, compared with default ARC. For 32 cores, ARC-TCC incurs 4.4X execution cycles and 11.7X on-chip traffic compared to ARC. TCC's mechanisms add high on-chip traffic by broadcasting write sets to all cores, and they incur high run-time overhead because many regions overflow the private caches, leading to much serialization. This comparison shows that, for the same context (i.e., precise consistency checking of SFRs), ARC's consistency mechanism provides substantial performance and traffic benefits over TCC's mechanism. Although follow-up work on TCC optimizes broadcast traffic (by adding a directory, making it closer to HTM designs that piggyback on coherence; Section 2.2) and parallelizes commits [32, 85], the fundamental bottleneck of *serialized execution of large regions* remains.

6.5 Summary

Our experiments up to 32 cores show that ARC compares favorably with both CE and TCC in terms of execution time and on- and off-chip traffic. ARC uses less on-chip and off-chip bandwidth than CE, which sometimes saturates the available on-chip bandwidth. The comparison with TCC's mechanism shows that ARC's mechanism is substantially more scalable and efficient than TCC's mechanism. Furthermore, ARC is competitive even with the MESI baseline in terms of time and traffic, while at the same time providing stronger consistency guarantees than MESI. Our evaluation thus shows ARC's value and viability.

7 Related Work

This section compares ARC with work *other than* the consistency mechanisms covered by Section 2.2.

Region serializability. Ouyang et al. *enforce* SFR serializability using a speculation-based approach that relies on extra cores to avoid substantial overhead [83]. *IFRit* detects conflicts between extended SFRs, but adds high run-time overhead and misses some SFR conflicts that compromise SFR serializability [38]. Other approaches support memory models based on serializability of *bounded* regions that are in general shorter than full SFRs [7, 30, 70, 74, 92, 97]. To provide end-to-end guarantees, these approaches require corresponding restrictions on compiler optimizations.

Read validation. Some software transactional memory (STM) systems use version or value validation of reads (e.g., [35, 52, 54, 82, 89]). To our knowledge, ARC's adaptation of validation to hardware (i.e., to the hierarchy of private and shared caches) and its combination of version and value validation are both novel.

Hardware support for detecting data races. Researchers have introduced custom hardware to accelerate data race detection, adding on-chip memory for tracking vector clocks and extending cache coherence [6, 37, 79, 91, 108, 111]. *Clean* simplifies race detection, but provides weaker consistency guarantees than SFRSx [91].

Rethinking cache coherence. Recently, there have been efforts to reduce the complexity of coherence protocols and the memory subsystem design by relying on disciplined parallelism [33, 103]. *DeNovo* and *DeNovoND* show that a simple coherence protocol can provide consistency for data-race-free (DRF) programs [33, 103]. In contrast, ARC provides consistency for *all* executions.

DeNovoSync, *SARC*, and *VIPS* optimize MOESI-style coherence protocols, using self-invalidation to reduce the cost and complexity of eager invalidation [60, 87, 102]. *TSO-CC* and *Racer* are coherence protocols that provide TSO using self-invalidation and without tracking sharers [40, 88]. Unlike ARC, these approaches provide consistency guarantees for data-race-free executions only. Jimborean et al. introduce compile-time analysis that safely extends SFRs based on a DRF assumption, reducing self-invalidation costs [58].

Some distributed shared memory (DSM) systems provide *release consistency*, which allows deferring coherence operations until synchronization operations for DRF programs [3, 12, 18, 29, 44, 61]. While ARC's coherence mechanisms are inspired by release consistency mechanisms, ARC's design provides strong consistency for all executions.

8 Conclusion

ARC is an architecture design that provides end-to-end SFRSx, ensuring strong, well-defined semantics for all executions. The key to ARC's efficiency is its novel consistency mechanism that allows regions to execute largely in isolation, deferring consistency and coherence to region boundaries and on evictions, without broadcasting access information or serializing cores' operations. ARC performs competitively with the state-of-the-art techniques CE [69] and TCC [50] in terms of run-time performance and on- and off-chip traffic, without incurring CE's high traffic costs or TCC's scalability bottlenecks. These results suggest that ARC advances the state of the art significantly in parallel architecture support for consistency mechanisms and strong consistency guarantees.

Acknowledgments

We thank Joe Devietti for sharing the RADISH simulator, and Nathan Beckmann for technical suggestions and feedback on the text.

9 References

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

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

[3] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *HPCA*, pages 26–37, 1996.

[4] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1996.

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

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

[7] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO*, pages 133–144, 2009.

[8] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In *DAMP*, pages 13–24, 2008.

[9] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA*, pages 316–327, 2005.

[10] L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *ISCA*, pages 115–126, 2008.

[11] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, pages 53–64, 2010.

[12] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, 1991.

[13] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, pages 72–81, 2008.

[15] S. Biswas, M. Cao, M. Zhang, M. D. Bond, and B. P. Wood. Lightweight Data Race Detection for Production Runs. In *International Conference on Compiler Construction*, pages 11–21, 2017.

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

[17] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13:422–426, 1970.

[18] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *ISCA*, pages 142–153, 1994.

[19] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *ISCA*, pages 24–34, New York, NY, USA, 2007.

[20] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA*, pages 127–138, 2008.

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

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

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

[24] H.-J. Boehm and S. V. Adve. You Don't Know Jack about Shared Variables or Memory Models. *CACM*, 55(2):48–54, 2012.

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

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

[27] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.

[28] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *PACT*, pages 187–200, 2014.

[29] M. Castro, P. Guedes, M. Sequeira, and M. Costa. Efficient and Flexible Object Sharing. In *ICPP*, August 1996.

[30] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, pages 278–289, 2007.

[31] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, pages 227–238, 2006.

[32] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *HPCA*, pages 97–108, 2007.

[33] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, pages 155–166, 2011.

[34] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, pages 258–269, 2002.

[35] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78, 2010.

[36] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, pages 85–96, 2009.

[37] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ISCA*, pages 201–212, 2012.

[38] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *OOPSLA*, pages 467–484, 2012.

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

[40] M. Elver and V. Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. In *HPCA*, pages 165–176, 2014.

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

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

[43] M. Eslamimehr and J. Palsberg. Race Directed Scheduling of Concurrent Programs. In *PPoPP*, pages 301–314, 2014.

[44] C. Fensch and M. Cintra. An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs. In *HPCA*, pages 355–366, 2008.

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

[46] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, pages 293–303, 2008.

[47] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-memory Multiprocessors. In *ASPLOS*, pages 245–257, 1991.

[48] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *ISCA*, pages 15–26, 1990.

[49] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An Exploratory Survey. In *EC*^{2}, 2008.

[50] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, pages 102–113, 2004.

[51] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

[52] T. Harris and K. Fraser. Revocable Locks for Non-Blocking Programming. In *PPoPP*, pages 72–82, 2005.

[53] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[54] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *PLDI*, pages 14–25, 2006.

[55] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[56] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two Hardware-Based Approaches for Deterministic Multiprocessor Replay. *CACM*, 52:93–100, 2009.

[57] Intel. Intel® Core^{TM} i7-3970X Processor Extreme Edition. `http://ark.intel.com/products/70845`.

[58] A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros. Automatic Detection of Extended Data-Race-Free Regions. In *CGO*, pages 14–26, 2017.

[59] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced Data Race Detection. In *SOSP*, pages 406–422, 2013.

[60] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54–65, 2010.

[61] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, pages 13–21, 1992.

[62] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *PPoPP*, pages 209–220, 2006.

[63] H. Labs. CACTI 5.3. `http://quid.hpl.hp.com:9081/cacti/`.

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

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

[66] C. Lin, V. Nagarajan, and R. Gupta. Efficient Sequential Consistency Using Conditional Fences. In *PACT*, pages 295–306, 2010.

[67] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, pages 273–286, 2012.

[68] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[69] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[70] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, pages 277–288, 2008.

[71] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, pages 190–200, 2005.

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

[73] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, 2009.

[74] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[75] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, pages 199–210, 2011.

[76] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *MICRO*, pages 3–14, 2002.

[77] A. Matveev and N. Shavit. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *ASPLOS*, pages 59–71, 2015.

[78] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

[79] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, pages 337–348, 2009.

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

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

[82] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *PACT*, pages 365–375, 2007.

[83] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

[84] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, pages 320–331, 2006.

[85] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *PACT*, pages 144–154, 2008.

[86] P. Ranganathan, V. Pai, and S. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *SPAA*, pages 199–210, 1997.

[87] A. Ros and S. Kaxiras. Complexity-Effective Multicore Coherence. In *PACT*, pages 241–252, 2012.

[88] A. Ros and S. Kaxiras. Racer: TSO Consistency via Race Detection. In *MICRO*, pages 1–13, 2016.

[89] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.

[90] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER Multiprocessors. In *PLDI*, pages 175–186, 2011.

[91] C. Segulja and T. S. Abdelrahman. Clean: A Race Detector with Cleaner Semantics. In *ISCA*, pages 401–413, 2015.

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

[93] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic Race Detection with LLVM Compiler. In *RV*, pages 110–114, 2012.

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

[95] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53(7):89–97, 2010.

[96] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *TOPLAS*, 10(2):282–312, 1988.

[97] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, pages 53–66, 2011.

[98] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-End Sequential Consistency. In *ISCA*, pages 524–535, 2012.

[99] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound Predictive Race Detection in Polynomial Time. In *POPL*, pages 387–400, 2012.

[100] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[101] C. SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. 1992.

[102] H. Sung and S. V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization Without Writer-Initiated Invalidations. In *ASPLOS*, pages 545–559, 2015.

[103] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. In *ASPLOS*, pages 13–26, 2013.

[104] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, pages 2–13, 2005.

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

[106] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, pages 115–128, 2003.

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

[108] B. P. Wood, L. Ceze, and D. Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*, pages 671–686, 2014.

[109] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA*, pages 261–272, 2007.

[110] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19:1–19:11, 2013.

[111] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, pages 121–132, 2007.