Rethinking Support for Region Conflict Exceptions

Swarnendu Biswas
Indian Institute of Technology
Kanpur, India
swarnendu@cse.iitk.ac.in

Rui Zhang
Ohio State University
Columbus, OH, USA
zhang.5944@osu.edu

Michael D. Bond
Ohio State University
Columbus, OH, USA
mikebond@cse.ohio-state.edu

Brandon Lucia
Carnegie Mellon University
Pittsburgh, PA, USA
blucia@cmu.edu

***Abstract*—Current shared-memory systems provide well-defined execution semantics *only for* data-race-free executions. A state-of-the-art technique called *Conflict Exceptions* (CE) extends M(O)ESI-based coherence to provide defined semantics to *all* program executions. However, CE incurs significant performance costs because of its need to frequently access metadata in memory.**

In this work, we explore designs for practical architecture support for region conflict exceptions. First, we propose an on-chip metadata cache called access information memory (AIM) to reduce memory accesses in CE. The extended design is called *CE*$^+$. In spite of the AIM, CE$^+$ stresses or saturates the on-chip interconnect and the off-chip memory network bandwidth because of its reliance on eager write-invalidation-based coherence. We explore whether detecting conflicts is *potentially* better suited to cache coherence based on release consistency and self-invalidation, rather than M(O)ESI-based coherence. We realize this insight in a novel architecture design called *ARC*.

Our evaluation shows that CE$^+$ improves the run-time performance and energy usage over CE for several applications across different core counts, but can suffer performance penalties from network saturation. ARC generally outperforms CE, and is competitive with CE$^+$ on average while stressing the on-chip interconnect and off-chip memory network much less, showing that coherence based on release consistency and self-invalidation is well suited to detecting region conflicts.

***Index Terms*—Conflict exceptions; data races; memory consistency models; region serializability**

I. Introduction

To maximize performance, shared-memory systems allow compilers and architectures to perform optimizations assuming that threads communicate only at synchronization operations. Consequently, a program that is *not* well synchronized permits *data races* and has complex or undefined semantics that lead to incorrect behavior [1].

Providing well-defined semantics for all programs, without restricting optimizations or impeding performance, is a long-standing challenge [1], [29], [30], [48]. A promising approach is for a shared-memory system to detect conflicts between executing *synchronization-free regions* (SFRs) [6], [31]. An SFR conflict generates a *consistency exception* because the conflict corresponds to a data race that may violate optimization assumptions. By detecting conflicts at regions demarcated only by synchronization operations,[1] the system provides a memory consistency model called *SFRSx*, in which program execution either appears to be a serialization of SFRs or terminates with a consistency exception, indicating a *true* data race [6], [31].

Supporting SFRSx demands precise conflict detection, requiring per-core tracking of byte-granular memory accesses. Prior work called *Conflict Exceptions* (CE) provides SFRSx with architecture support [31]. However, CE frequently exchanges metadata between cores and memory on private cache misses and evictions, leading to high bandwidth requirements that can potentially saturate the on-chip interconnect and off-chip memory, thereby degrading performance (Sections II and VI).

This work proposes designs for architecture support for *efficient* region conflict detection. The *first* contribution is to try to address CE's high performance overheads resulting from frequent memory accesses by adding a metadata cache adjacent to the shared last-level cache (LLC), called the *access information memory* (AIM). We call the resulting architecture *CE*$^+$. Despite the optimization, CE$^+$ still incurs significant costs in order to support precise conflict detection *eagerly* at every memory access. CE and CE$^+$ detect conflicts eagerly by relying on an eager write-invalidation-based coherence protocol like M(O)ESI [49] to exchange access metadata between concurrently executing cores, which can saturate the on-chip interconnect and the off-chip memory network.

The *second* contribution of this work is a novel architecture design called *ARC* that, like CE and CE$^+$, provides SFRSx, but substantially differs in *how* it supports conflict detection and cache coherence. The key insight of ARC is that, while M(O)ESI coherence suits a data-race-free (DRF) assumption, region conflict detection is potentially better served by a "lazier" approach to coherence. ARC provides unified mechanisms for cache coherence and conflict detection by extending *release consistency* and *self-invalidation* mechanisms, which prior work has employed, but for coherence alone and assuming DRF [11], [19], [26], [27] (Section VII). Like ARC, *Transactional Coherence and Consistency* (TCC) unifies coherence and conflict detection, but relies on broadcast and serialized transaction commit [20], incurring high costs (Section VI-C).

This material is based upon work supported by the National Science Foundation under Grants XPS-1629126, CCF-1421612, CAREER-1253703, and CSR-1218695.

[1]In contrast, *DRFx* detects conflicts between *bounded-length* regions, requiring restrictions on compiler optimizations [34], while other proposals detect every data race and incur high costs [16], [53].

CE$^+$ and ARC place different demands on the existing microarchitecture. Whereas CE$^+$ extends CE and hence *relies* on support for M(O)ESI cache coherence, ARC *does not* depend on a M(O)ESI implementation or directory and does not require a core-to-core interconnect. Like CE, both CE$^+$ and ARC add per-byte *access information* to caches and require a backing memory store for evicted access information. An access information cache in CE$^+$ and ARC avoids memory lookups at an acceptable area and power overhead, and ARC adds distributed *consistency controller* logic (Section V-C). CE$^+$ and ARC target widely available CMPs with moderate core counts ($\leq$32). CMPs with large core counts (>32) are out of scope because access information storage requirements scale with core count.

We evaluate CE$^+$ and ARC and compare them with CE. CE$^+$ consumes less memory bandwidth, improving performance and energy compared to CE for a number of applications and core counts, which shows the potential for the AIM. However, CE$^+$ can still suffer from performance penalties from network saturation. ARC also outperforms CE, and has comparable run-time performance and energy usage on average with CE$^+$. In general, ARC's lazy approach to coherence means it has less on-chip and off-chip bandwidth requirements, benefiting applications with large regions and working set sizes (Section VI-B). We also compare ARC with TCC [20] adapted to provide SFRSx. ARC avoids communication and serialization issues faced by CE, CE$^+$, and TCC. Furthermore, we show that ARC achieves well-defined semantics at modest overheads compared to current shared-memory systems that provide *weak* memory consistency (Section VI-D). Our results show that CE$^+$ and ARC advance the state of the art in architecture support for well-defined memory semantics.

II. Background: SFRSx and Conflict Exceptions

As the prior section motivated, a system can provide well-defined execution semantics by detecting conflicts between synchronization-free regions (SFRs). An SFR is a sequence of non-synchronization instructions executed by a single thread, delimited by synchronization operations (e.g., lock acquire, lock release, thread fork, and thread join), as shown in Figure 1. If a system detects conflicts between unbounded SFRs and generates a consistency exception upon a detected conflict, it provides the *SFRSx* memory consistency model, which ensures either SFR serializability or a consistency exception indicating a data race [6], [31]. SFRSx thus ensures strong, well-defined semantics for *all* programs.

Providing SFRSx is different from detecting all data races. Figure 1 shows an execution with data races on variables x and y. Under SFRSx, a system does *not* need to generate a consistency exception at the read of x for *this observed execution* because the SFRs accessing x do not overlap. In contrast, the SFRs accessing y overlap; SFRSx throws a consistency exception if it cannot ensure rd y's SFR serializes before or after wr y's SFR. Note that an execution may or may not generate an exception at each of Thread 2's (racy)

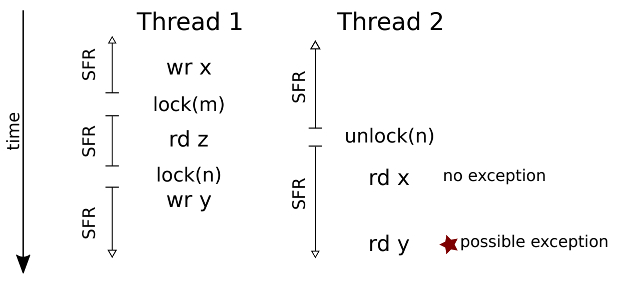


Fig. 1. Under SFRSx, an execution generates a consistency exception for a data race that may violate SFR serializability.

accesses under SFRSx; if the execution does not generate an exception, it must preserve SFR serializability.

Detecting region conflicts: Prior architectures for both precise and imprecise conflict detection of unbounded regions have the key limitation that they consume a large amount of on-chip bandwidth or increase traffic to main memory [20], [31], [35]. The rest of this section summarizes work that uses *precise* conflict detection, which is required to provide SFRSx.

Conflict Exceptions (CE) provides SFRSx, adding per-byte access metadata to each cache line to track reads and writes [31]. CE adds local and remote access bits for each byte in a private line to keep track of bytes read or written by an ongoing region in local or remote cores, respectively. CE piggybacks on MOESI coherence [49] to exchange metadata indicated by the per-byte access bits, and detects SFR conflicts by comparing local and remote access bits. For any conflict detected, CE generates a consistency exception to terminate the execution. A core sends its local access bits in an end-of-region (endR) message to other cores at each region boundary. A core receiving an endR message clears the corresponding remote bits from its private caches and sends back an acknowledgment. CE also handles evictions from private caches to the LLC by storing local access bits of evicted lines in a per-process structure called the *global table*. Communication at region boundaries and frequent access of the *in-memory* global table often lead to saturating the on-chip interconnect and off-chip memory bandwidth (shown empirically in Section VI-B).

III. Optimizing Conflict Exceptions with the AIM

Our first contribution in this paper is an optimized design of CE, called *CE$^+$*. CE$^+$ includes a dedicated cache structure, called the *access information memory* (AIM), for storing access information that are evicted from private caches. CE$^+$ aims to reduce expensive accesses to the *in-memory* global table in CE by backing metadata for lines evicted from private caches in the AIM. Conceptually, the AIM sits adjacent to the LLC, and the global table is accessed *only* on AIM misses. An AIM entry corresponds to an LLC line, storing for each byte one read bit for each of the C cores, and the current writer core if there is one (a $\lg C$-bit writer core and 1 bit indicating there is a writer). As shown in Figure 2 (ignore the portion shaded gray), with B-byte cache lines, an AIM

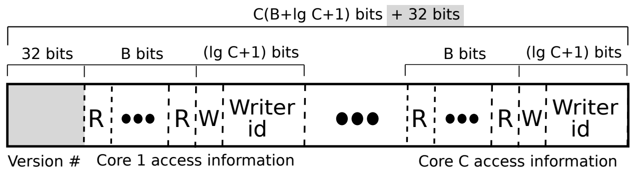


Fig. 2. An AIM entry for a processor with C cores and B-byte cache lines. The shaded portion represents version information, which is not required by CE$^+$ but is part of ARC (Section V-B).

entry is $(C + 1 + \lg C) \times B$ bits, e.g., 96 bytes for 8 cores and 64-byte lines.

As a centralized structure, contention by cores accessing the AIM threatens scalability at high core counts. Address-banking the AIM reduces contention, and mitigates the threat to scalability; we assume 8 AIM banks.

An ideal AIM with one entry per LLC line is a perfect cache of the LLC's access information. However, an ideal AIM is impractical. With 8 cores, 64-byte lines, and a 16 MB, 16-way LLC, the AIM would be around 24 MB—a large (59.2 mm^2), slow (9 ns access time), and power-hungry (394 mW leakage per bank) on-chip structure in 32-nm technology (data from CACTI 7 [24]). Instead, CE$^+$ uses a realistic AIM design that, for 8 cores, has 32K entries and 4-way associativity. This 3 MB AIM has implementable area (9.2 mm^2), latency (3.3 ns access time), and leakage (52.7 mW per bank). In a 6-core Intel Core i7-3970X at 3.5 GHz,[2] this AIM would add 0.39% overhead to the 2,362 mm^2 package area, 12-cycle access latency, and negligible leakage of 0.3% of the thermal design power (TDP).

The AIM's hardware design scales well across a moderate range (up to 32) of CMP core counts. At 16 cores, a 32K-entry AIM is 5.3 MB, has 4.6 ns access time, 13.5 mm^2 area, and 89 mW leakage per bank. At 32 cores, a 64K-entry AIM is of 19 MB size, with 7.8 ns access time, 53.8 mm^2 area, and 310 mW leakage per bank. We choose these AIM sizes to balance AIM misses and hardware cost.

Since AIM size, latency, and leakage scale with core count, the AIM is unlikely to scale to large (> 32) core counts. At 64 cores, a 128K-entry AIM would be 71 MB with 13.8 ns access time, 162.8 mm^2 area, and 1108 mW leakage per bank. In the Intel Core i7-3970X, such an AIM would incur around 6% area overhead, 49-cycle access latency, and substantial leakage of 6.0% of the TDP; such a structure is too costly to implement. Using fewer (e.g., 64K) entries decreases area and power costs, but increases the AIM's miss rate, increasing a computation's latency and total energy consumption.

IV. Design Overview of ARC

The second contribution of this work is to present a new architecture called *ARC* (Architecture for Region Consistency) that provides the SFRSx memory model. Unlike CE and CE$^+$, which rely on M(O)ESI coherence, ARC exploits synergy between (1) conflict detection and (2) coherence based on release consistency and self-invalidation. In *release consistency*, a core's private cache waits to propagate writes until a synchronization *release* operation [19], [27]. In *self-invalidation*, a core invalidates lines cached privately that may be out of date at synchronization *acquire* operations [11], [26]. In contrast with M(O)ESI coherence, release consistency and self-invalidation do *not* eagerly invalidate lines when they are written by another core. Such "lazy" invalidation allows an ARC core to execute regions mostly in isolation, performing coherence and conflict detection only at region boundaries (synchronization operations) and on evictions to the shared cache. ARC's *novel* approach for committing writes and *version- and value-validating* reads minimizes communication required for detecting conflicts and avoids most self-invalidation costs.

ARC requires minimal compiler support to identify synchronization operations, which serve as region boundaries. ARC does not restrict compiler optimizations within regions.

The rest of this section overviews ARC's design. Section V describes an architecture design that implements ARC.

State: ARC supports byte-granular tracking to provide precise conflict detection required by SFRSx. Cores' private caches and the 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. To help validate reads and limit self-invalidation, each LLC line maintains a *version*, which is a monotonically increasing integer that represents the latest write-back to the line in the LLC, and is incremented each time the line is written back to the LLC.

Actions at reads and writes: When a core reads (writes) a byte of memory, it updates the read (write) bit for the accessed byte in its private cache. If the byte was previously written and is being read, the core does not update the read bit. Aside from fetching a line from the LLC on a private cache miss, a read or write does *not* trigger any communication with the LLC or other cores.

A. Actions at Region Boundaries

When a core's region ends, it provides both coherence and SFR serializability using a *region commit protocol.* Unlike other mechanisms (e.g., TCC [20]; Section VI-C), the region commit protocol for a core can proceed *in parallel* with other cores performing the protocol or executing regions, because the core and the LLC do *not* communicate with other cores' caches during the protocol. The protocol ensures atomicity by setting a core's write access bits in the LLC for the duration of the protocol. The protocol consists of the following three operations in order:

(1) *Pre-commit:* For each dirty line, the core sends its privately cached *write* bits and *version* to the LLC. The LLC checks for any conflicting access bit for the same byte in the LLC, which indicates a conflict. If the version matches the LLC line's

[2]http://ark.intel.com/products/70845

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

version, then the core's cached line is up to date and is not invalidated during post-commit (described below). Otherwise, the LLC sends a "must invalidate line" message to the core.

(2) *Read validation:* The core must validate that the values it read are consistent with the LLC's current values. Instead of sending each line's data, which would be expensive, ARC sends the line's *version* to the LLC. The LLC compares the line's version with *its* version of the line. A successful *version validation* of the line implies the core read valid values during the region. On a version mismatch, which indicates a potential conflict, the LLC responds with its version and data values for the line. The core *value-validates* the line precisely by comparing the LLC's values with its cached values (looking only at bytes with read bits set) and generates a consistency exception if they do not match; otherwise the core updates its version and values. Even on a version match, if *any* write bit is set in the LLC line for a remote core, the LLC responds with its line's write bits. The core ensures the absence of a write–read conflict by checking that no locally read byte has its write bit set in the LLC. To ensure validation against a consistent LLC snapshot, read validation must repeat until it validates every line without any version mismatches. Starvation is possible if a core repeatedly retries read validation, but ARC is *livelock and deadlock free*: a version mismatch implies that some other core made progress by writing to the LLC. A misbehaving thread of a process P can starve other threads of process P only. P's misbehaving thread *cannot* starve another process, Q (which would be a denial-of-service attack), because P and Q access distinct lines (assuming no interprocess sharing).

A long-running region should occasionally validate its reads, to detect a data race that causes a region to get stuck in an infinite loop that is infeasible in any SFR-serializable execution (a so-called "zombie" region [21]).

Read validation ensures SFR serializability, even considering the *ABA problem*, because it validates values against a consistent LLC snapshot: it ensures that a core's read values match values in the LLC (and no conflicting write bits are set) at the point in time when read validation (re)started. If ABA happens, read validation will detect a version mismatch but not a value mismatch, update the private line version, and retry.

A core c can skip validating any line that was not updated in the LLC by any other core during c's region. ARC maintains a per-core *write signature* at the LLC that encodes *which* lines have been updated in the LLC during the core's current region *by any other core*. The core receives the write signature from the LLC at the start of read validation. It re-fetches the write signature from the LLC at the end of read validation to ensure that it has not changed; if it has, read validation restarts.

(3) *Post-commit:* The core writes back dirty bytes to the LLC (these write-backs can be deferred; Section V-C1) and clears its private cache's access information. The LLC clears all of its access information for the core. By keeping write bits set from pre- to post-commit, ARC ensures that commit and validation appear to happen together atomically.

In a naïve design, the core must then invalidate all lines in its private cache. However, a core can avoid invalidating most lines by leveraging ARC's existing mechanisms. A core can avoid invalidating most lines *accessed by the ending region* because read validation has already ensured that read-only lines are up-to-date with the LLC, and pre-commit has ensured that written-to lines are up-to-date with the LLC, except for lines for which the LLC sends a "must invalidate line" message to the core. A core can avoid invalidating a line *not* accessed by the ending region if ARC can ensure that other cores have not written to the line in the LLC during the region's execution, identified using the same per-core write signature that read validation uses.

B. Evictions and WAR Upgrades

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 already evicted the same line, or that later 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.

When a core writes a byte that it read earlier in its ongoing region—called a *write-after-read (WAR) upgrade*—the private cache cannot simply overwrite the byte because that would make it impossible to value-validate the prior read. ARC instead immediately sends a WAR-upgraded line's read bits and version to the LLC. The LLC read-validates the line and detects future read–write conflicts for the line, similar to how private cache line evictions are handled. As the next section explains, the ARC architecture avoids communicating with the LLC for WAR-upgraded L1 lines, by preserving an unmodified copy of the line in the L2.

V. Architecture Design of ARC

The ARC architecture is a collection of modifications to a baseline multi-core processor. The cores share the last-level cache (LLC), and each core has a cache hierarchy with private, write-back L1 and L2 caches. Unlike CE and CE$^+$, ARC need not assume that the LLC is inclusive of a core's private L1 and L2 caches. ARC's baseline processor has *no support for cache coherence*: it has no directory, and each cache line has only a *valid* bit and a *dirty* bit. Figure 3 shows the components (shaded blocks) that ARC adds to the baseline processor: (1) access information storage and management and (2) distributed per-core consistency controllers (CCs), each discussed next.

A. Private Access Information Management

Every L1 and L2 cache line maintains access information and a 32-bit version (as shown in Figure 4) that ARC uses to detect conflicts. ARC associates a read bit and a write bit per byte with each line in the core's L1 and L2 caches.

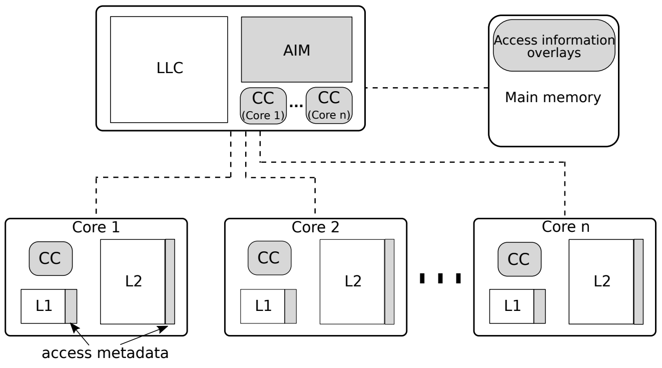


Fig. 3. The ARC architecture (not according to scale). Hardware structures added by ARC are shaded.

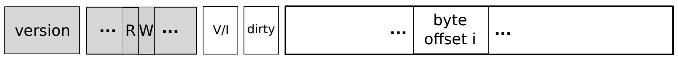


Fig. 4. Per-line metadata introduced by ARC for private caches. Metadata added by ARC is shaded.

Updating access information: When a core reads (writes) a byte, it sets the byte's read (write) bit if it is not already set. At reads, the read bit is not set if the write bit is already set. For a WAR-upgraded L1 line, ARC relies on the original value's presence in the L2, and copies its access information to the L2. If the WAR-upgraded L1 line is later evicted, ARC validates the L2 line immediately, as Section V-C2 describes.

Evictions: When a core evicts a line from L1 to L2, the line's access information is copied to an identical bit array for the line in the L2. When the L2 evicts a line, the core sends the line's access information to the LLC's *access information memory* (AIM), as described later in this section.

B. LLC Access Information Management

Like CE$^+$, ARC stores access information for the LLC in the AIM (Section III). In addition to read and write bits equivalent to CE$^+$'s, an AIM entry in ARC contains a 32-bit version that is used during read validation (shaded portion in Figure 2).

When a core writes back a line to the LLC, the core's AIM-side CC (described next) updates the line's AIM entry to reflect the line's access information. When a core writes back a dirty line to the LLC, the core's AIM-side CC increments the line's version in the AIM. A line's version is only incremented by the AIM, never by a core.

ARC must preserve the access information for evicted AIM entries. ARC augments an evicted AIM entry with a list of *epochs*, one per core, before sending an entry to memory. An epoch is a number that identifies a core's ongoing SFR, and the AIM stores each core's epoch in a dedicated *current epoch register*. The AIM increments a core's epoch when the core finishes an SFR. On a fill, the AIM compares the entry's epochs to each core's epoch. A differing epoch indicates access information from a completed SFR, allowing the AIM to lazily clear information for that line for that core. The epoch list avoids the need to explicitly track which lines in memory have access information.

With C cores, B-byte cache lines, 4-byte versions, and E-bit epochs, each line evicted from the AIM occupies $4 + \lceil (C + 1 + \lg C) \times B + E \times C) \times \frac{1}{8} \rceil$ bytes of memory. ARC uses *page overlays* [45] to consume memory only for lines that have AIM-evicted access information. Page overlays extend the TLB and page tables to provide efficient, compact access to a page's per-line metadata.

C. Consistency Controllers (CCs)

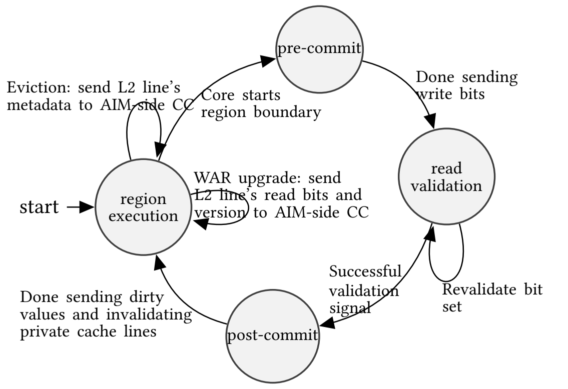
Section IV described the steps of ARC's *region commit protocol*. Here we focus on the implementation of the *consistency controllers* (CCs), which contain buffering and control logic for exchanging access bits, versions, and values. Each core has a *core-side* CC and an *AIM-side* CC. The CCs themselves are unlikely to limit scalability because different cores' CCs share no state or control logic at the core or AIM side. Contention at the AIM by different cores' AIM-side CCs is unlikely to limit scalability because the AIM is banked and accesses to it are infrequent relative to region execution.

1) Region Commit Protocol: Figure 5 shows the high-level states and transitions of a core's core-side and AIM-side CCs. During region execution, the core-side and AIM-side CCs exchange access information to handle WAR upgrades and evictions. The core-side and AIM-side CCs also coordinate during the other protocol phases. The figure omits transitions for consistency exceptions to avoid clutter; ARC may deliver consistency exceptions for a conflict detected during execution, pre-commit, or read validation.

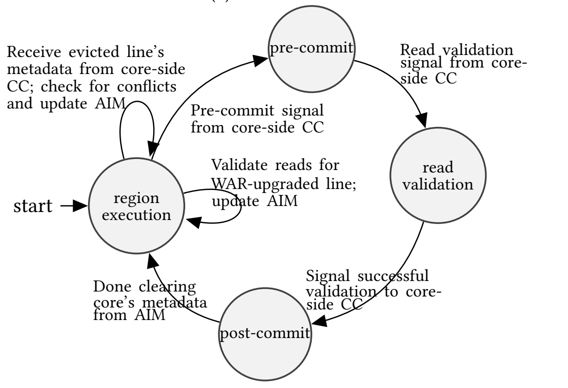
A core's core-side CC initiates the commit protocol. A core's protocol phases can *overlap with other cores* performing the protocol or executing regions; the protocol ensures atomicity by setting a core's write bits in the AIM during pre-commit and not clearing them until post-commit. Different cores' CCs never communicate directly; instead, a core's CC checks consistency using only data in the LLC and metadata in the core's cache and in the AIM.

Pre-commit: The core sequentially sends write bits from its dirty cached lines to its AIM-side CC. The core's AIM-side CC compares the core's write bits to all other cores' access bits from the AIM to check for a conflict. Upon a conflict, the core's CC delivers an exception to the core. If there is no conflict, the AIM-side CC updates the AIM entry's access bits to match the buffered ones received from the core-side CC.

Read validation: The core-side CC performs read validation, sending a sequence of validation request messages to its AIM-side CC, one for each line the core read during the ending region (lines 1–7 in Figure 6). Each message contains the line address and version from the core's private cache. For each message it receives, the AIM-side CC compares with 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-side CC with write bits so the core-side CC can check for write–read conflicts. In case of a conflict, the core raises a consistency exception.


pre-commit
Done sending write bits
read validation
Revalidate bit set
Successful validation signal
post-commit
Done sending dirty values and invalidating private cache lines
region execution
start
Eviction: send L2 line's metadata to AIM-side CC
Core starts region boundary
WAR upgrade: send L2 line's read bits and version to AIM-side CC

(a) Core-side CC


pre-commit
Read validation signal from core-side CC
read validation
Signal successful validation to core-side CC
post-commit
Done clearing core's metadata from AIM
region execution
start
Receive evicted line's metadata from core-side CC; check for conflicts and update AIM
Pre-commit signal from core-side CC
Validate reads for WAR-upgraded line; update AIM

(b) AIM-side CC

Fig. 5. State diagrams for a core's core- and AIM-side CCs.

```
1: repeat
2:     $revalidateBit \leftarrow$ false
3:     for all private cache lines $L$ with a read-only byte do
4:         Send $L$'s address and version to AIM-side CC
5:     end for
6:     Wait until AIM-side CC signals done validating last line
7: until not $revalidateBit$
       ▷ Handler for asynchronous responses:
8: for all responses $\langle a, v', w', d' \rangle$ from AIM-side CC do
       ▷ Response is LLC line's address, version, write bits, & data values
9:     $revalidateBit \leftarrow$ true
10:    $d \leftarrow getValues(L)$                ▷ Get privately cached data values
11:    if $d' \neq d \vee w' \cap readBits(L) \neq \emptyset$ ▷ Compare read-only bytes only
12:        then Consistency exception!
13:    $setVersion(L, v')$
14:    $setValues(L, d')$
15: end for
```

Fig. 6. Details of core-side CC's *read validation* state.

If a line's version mismatches, another core wrote to the line and there may be a conflict. On a version mismatch, the core's AIM-side CC logic sends the core-side CC the line's updated version. Lines 8–15 of Figure 6 show how the core-side CC handles responses from the AIM-side CC. The core re-fetches that line from the LLC into a dedicated *line comparison buffer* in the core. The core-side CC compares the (read-only) line in the private cache to the line in the comparison buffer. If the lines differ for a byte that has its read bit set, 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; the core sets its *revalidate bit* because the core must *revalidate* prior lines. After a core finishes validating all remaining lines, it revalidates by starting again from the beginning, streaming versions for comparison with the AIM by the AIM-side CC. After the core completes validation (or revalidation) without version mismatches, it unsets the revalidate bit and continues.

Post-commit: The core-side CC clears L1 and L2 lines' access bits, and the AIM-side CC clears the core's access bits in the AIM and finally increments the core's epoch.

Instead of writing back dirty L1 and L2 bytes to the LLC, ARC optimizes post-commit by *deferring* write-back of each dirty line until another core needs it. Deferring write-backs adds $1 + \lg C$ bits (for a system with C cores) per LLC line to identify whether its state is *deferred* and which *last-writer* core has the deferred data. Writing back the write bits is required to allow the CC to detect conflicts and commit writes atomically. If another core requests a deferred line from the LLC, the LLC first fetches the latest values from the last-writer core before responding to the request.

The core-side CC invalidates L1 and L2 lines that cannot be kept valid based on pre-commit, read validation, or the per-core write signature. As an optimization, private cache lines have a special *cond-invalid* (CINV) state in addition to *valid* (V) and *invalid* (I) states; CINV indicates that the line's data is valid *only if* the LLC's version is unchanged. During post-commit, a core optimistically changes each untouched line's state to CINV, instead of I. When a core accesses a line in the CINV state for the first time in a subsequent region, the core's CC sends its copy of the line's *version* to the AIM-side CC, which compares the version with the AIM's version and replies to the core indicating whether the versions match. If the versions match, the core-side CC upgrades the line in the L2 and L1 caches to V. Otherwise, the access is handled as a miss.

2) Other CC Responsibilities: Besides performing the region commit protocol, the core- and AIM-side CCs have the following responsibilities.

Per-core write signatures: The AIM-side CCs encode the write signature for each core as a Bloom filter. Whenever a core writes back to the LLC, its AIM-side CC updates *every other* core's write signature to include the updated line. When a core starts read validation, the core's AIM-side CC sends the core-side CC its write signature and clears the AIM-side CC's copy of the core's signature. The core-side CC uses its received copy of the write signature during read validation to identify lines that do not need to be validated, and during post-commit to identify lines that do not need to be invalidated. 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 VI-A).

Handling evictions and WAR upgrades: When an L2 evicts a line with access information, the core's AIM-side CC performs pre-commit and read validation on the line. Likewise, if an L2 sends access information for a WAR-upgraded L2 line (Section V-A), the AIM-side CC performs read validation on the line. The AIM-side CC checks for conflicts using the access information in the AIM, and checks that the L2 line's contents match the version or, if the versions do not match, 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 a conflict, it generates a consistency exception, by raising a non-maskable interrupt signal for the core that detected the conflict. The core receives the interrupt and then runs operating system code to terminate the execution.

D. Other Issues

Implementing synchronization: CE and CE$^+$ support lock-based synchronization using M(O)ESI. By forgoing M(O)ESI coherence, ARC needs a special mechanism to implement lock acquire and release. ARC uses a mechanism similar to distributed queue locks used by *DeNovoND* [52]; alternatively, *callbacks* could efficiently implement locks with self-invalidation [39]. We assume compiler support to identify synchronization as region boundaries (e.g., endR instruction in CE [31]). ARC can handle legacy code by intercepting pthread calls to identify them as synchronization, but a library approach alone does not support other synchronization strategies (e.g., atomics and assembly).

Handling context switches and translation shootdowns: To avoid false conflicts, thread migration in ARC is allowed only at synchronization points. Furthermore, the region commit protocol needs to complete before a thread is migrated, to avoid missing conflicts.

A core can context-switch from one process's thread to *another process's* thread at any time (assuming no interprocess memory sharing), which preserves the operating system's process scheduling behavior. A core can only switch from one thread to another thread *from the same process* at a synchronization point to avoid missing conflicts between the threads. If a swapped-in thread evicts a privately cached line accessed by a swapped-out thread, the eviction may lead to a consistency exception. The operating system can use the page table to identify the process that originally set the metadata bits, and deliver the exception to that process. Page re-mapping (e.g., changing page permissions) can flush access bits with the TLB shootdown to avoid future false conflicts on the re-mapped page, or re-mapping could end the current region.

VI. Evaluation

This section evaluates run-time performance and energy usage of CE$^+$ and ARC, compared primarily with CE [31]. We also compare ARC with TCC's mechanisms [20] and with a contemporary shared-memory system that provides weak execution guarantees in the presence of data races.

Processor		4-, 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 (for CE and CE$^+$)
LLC		64 B line size
	4 cores:	8-way 8 MB shared cache, 25-cycle hit latency
	8 cores:	16-way 16 MB shared cache, 35-cycle hit latency
	16 cores:	16-way 32 MB shared cache, 40-cycle hit latency
	32 cores:	32-way 64 MB shared cache, 50-cycle hit latency
AIM cache		4-way metadata cache with 8 banks
CE$^+$	4 cores:	56 B line size (~1.8 MB), 32K lines, 4-cycle hit latency
	8 cores:	96 B line size (~3 MB), 32K lines, 6-cycle hit latency
	16 cores:	168 B line size (~5.3 MB), 32K lines, 10-cycle hit latency
	32 cores:	304 B line size (~19 MB), 64K lines, 15-cycle hit latency
ARC	4 cores:	60 B line size (~1.9 MB), 32K lines, 4-cycle hit latency
	8 cores:	100 B line size (~3.2 MB), 32K lines, 6-cycle hit latency
	16 cores:	172 B line size (~5.4 MB), 32K lines, 10-cycle hit latency
	32 cores:	308 B line size (~19.3 MB), 64K lines, 15-cycle hit latency
Memory		120-cycle latency
Bandwidth		NoC: 100 GB/s, 16-byte flits; Memory: 48 GB/s

TABLE I
Architectural parameters used for simulation.

A. Simulation Methodology

We implemented CE, CE$^+$, and ARC in simulation. The simulators are Java applications that implement each architecture; they use Pin [32] to generate a serialized event trace that the simulators consume. For each program execution and core count, all simulators run the same serialized event trace from Pin to eliminate differences due to run-to-run nondeterminism. We send simulation output data to McPAT [28] to compute energy usage. The CE and CE$^+$ simulators extend the directory-based MESI cache coherence protocol implemented in the RADISH simulator, provided by its authors [14]. We have made our Pin frontend and CE, CE$^+$, and ARC simulator backends publicly available.[4]

Table I shows simulation parameters for 4–32 cores. CE and CE$^+$ use an LLC that is inclusive, to support MESI with the directory embedded in the LLC (see Figure 8.6 in [49]). ARC's LLC is *not* inclusive (Section V). The simulators treat pthreads calls as lock operations. ARC treats atomic accesses (i.e., those with the x86 LOCK prefix) as special, handling them like locks (Section V-D) that do not delineate regions.

Modeling execution costs: We use an idealized core model with an IPC of one for non-memory instructions. Table I shows instruction cycle costs. Our simulators report the maximum cycles for any core; as in prior work [5], [14], the simulators do not model synchronization wait time. We model wait-free, write-back caches with idealized write buffers. Our simulation ignores the effects of context switching and page remapping. We compute energy usage using the McPAT modeling tool [28].

[4]https://github.com/PLaSSticity/ce-arc-simulator-ipdps19

McPAT takes as input architectural specifications and dynamic statistics corresponding to an execution (e.g., cache misses, coherence events, and simulated execution cycles), and reports static and dynamic power usage of specified architectures. Since our simulators do not collect core-level statistics such as ALU and branch instructions, our methodology uses McPAT to compute power consumption for the cache and memory subsystem only, including the on-chip interconnect and LLC-to-memory communication, and computes corresponding energy usage.

To model the costs of ARC's operations at region boundaries, when cores send messages without synchronous responses during pre-commit and read validation, we compute the cycle cost of messages based on the total message size and bandwidth between a core and the LLC. The ARC simulator models the full cost of version mismatches during read validation, including *repeated* validation attempts. However, since the simulators process a serialized event trace, the number of read validation attempts for a region cannot exceed two in our evaluation.

We simulate an interconnect with 16-byte flits and with bandwidth characteristics as shown in Table I. A control message is 8 bytes (tag plus type); a MESI data message in the CE simulators is 64 bytes (i.e., a cache line). For ARC write-backs, we model idealized write-buffer coalescing that sends only dirty bytes. When sending versions to the LLC during read validation, each flit holds four lines of data; we assume that the core's AIM-side CC and the LLC are ported to handle a message's four validation requests.

Our simulators compute the bandwidth required by the different techniques by tracking the amount of data that is transmitted on the on-chip interconnect and the off-chip memory network during application execution. To keep the complexity of the simulators manageable, the simulators do not model queuing in the on-chip and off-chip networks. We approximate the effects of queuing by scaling execution cycles by the proportion with which the assumed on-chip and off-chip bandwidths (Table I) are exceeded. This simple methodology *does not* model network stalls due to bursts of traffic saturating network-internal buffers. For example, ARC may suffer periodic bursts in bandwidth consumption and network stalls while executing the region commit protocol, as can CE and CE$^+$ when broadcasting the endR message at region boundaries.

Benchmarks: Our experiments execute the PARSEC benchmarks [5], version 3.0-beta-20150206, with simmedium inputs. We omit freqmine since it uses OpenMP as the parallelization model, and facesim since our Pintool fails to finish executing it. We measure execution cost for the parallel "region of interest" (ROI) only; vips lacks an ROI annotation so we use its entire execution as the ROI. Table II shows how many threads each benchmark spawns, parameterized by n, which is PARSEC's *minimum threads* parameter. The simulators set n equal to the number of cores (4, 8, 16, or 32) in the simulated architecture. The last three columns show the average number of memory accesses performed per SFR. The simulators map threads to cores using modulo arithmetic.

		Average accesses per SFR ($\times 10^3$)			
	Threads	$n = 4$	$n = 8$	$n = 16$	$n = 32$
blackscholes	$1 + n$	19,100	9,540	4,770	2,380
bodytrack	$2 + n$	52.9	47.7	40	30.1
canneal	$1 + n$	522	261	131	65.2
dedup	$3 + 3n$	42.9	43.2	43.1	42.6
ferret	$3 + 4n$	914	811	626	465
fluidanimate	$1 + n$	0.215	0.116	0.086	0.056
raytrace	$1 + n$	9,710	5,230	2,720	1,390
streamcluster	$1 + 2n$	4.65	1.36	0.407	0.122
swaptions	$1 + n$	165,000	82,500	41,300	20,600
vips	$3 + n$	122	102	76.2	50.8
x264	$1 + 2f$	220	205	198	198

TABLE II
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.

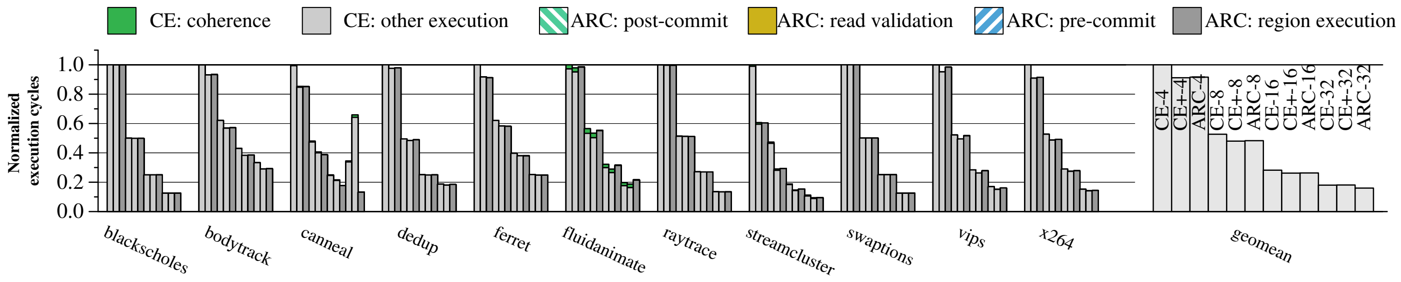
Consistency exceptions: When the CE, CE$^+$, and ARC simulators detect conditions for a consistency exception, they log the exception and continue execution. In our experiments, CE/CE$^+$ and ARC detect conflicts in canneal and streamcluster, and CE/CE$^+$ also detects conflicts in vips. These differences arise because CE/CE$^+$ detects all conflicts eagerly, while ARC detects some conflicts lazily. The simulators can report both locations involved in a conflict, by maintaining the last-access source location corresponding to each read and write bit of access information. Using Google's ThreadSanitizer [44] and by implementing "collision analysis" [18], we have confirmed that each detected conflict corresponds to a true data race.

B. Performance and Energy Comparison

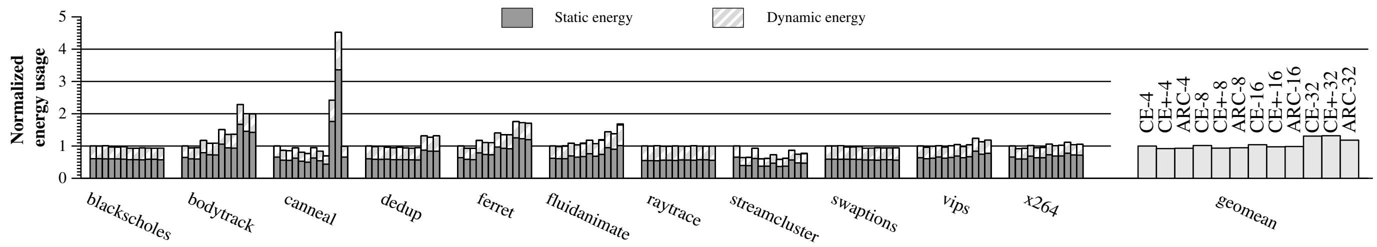
Figure 7 shows our main results. The configurations (*CE-4*, *CE+-4*, *ARC-4*, etc.) show the run-time performance and energy usage for CE, CE$^+$, and ARC on 4, 8, 16, and 32 cores, respectively, normalized to CE-4.

Figure 7(a) shows executed cycles as reported by the simulators, broken down into different components. CE and CE$^+$ are divided into cycles attributed to MESI *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 7(b) compares the energy usage of CE, CE$^+$, and ARC. Each bar shows the breakdown of total energy consumed into energy due to static and dynamic power dissipation, as computed by McPAT. The static and dynamic energy components for CE$^+$ and ARC include contributions from using the AIM cache, and the dynamic component for ARC includes the contribution from Bloom filter accesses.

Figures 7(a) and 7(b) show that CE$^+$ improves run-time performance and energy usage over CE across core counts for the majority of programs. For 4, 8, and 16 cores, CE$^+$ improves execution cycles and energy usage over CE by 8.8% and 7.9%, 8.9% and 7.9%, and 7.1% and 6.2%, respectively. CE backs up access bits in an *in-memory* table when a line that



(a) Execution time. Time spent in ARC's region commit protocol is often too small to be visible.



(b) Energy usage.

Fig. 7. Execution time and energy usage for CE, CE$^+$, and ARC for 4–32 cores, normalized to CE with 4 cores (*CE-4*).

was accessed in an ongoing region is evicted from a private cache. CE accesses memory even on an LLC hit, for a line that was previously evicted from a private cache or the LLC during the ongoing region. The on-chip AIM in CE$^+$ helps avoid some of CE's expensive memory accesses. Overall, these results show promise in using a metadata cache to reduce off-chip memory traffic, thus improving performance and saving energy.

On 32 cores, CE$^+$ fares the same as or better than CE for all programs except canneal. canneal has large regions and working sets, which leads CE to saturate off-chip memory bandwidth (120 GB/s compared to 48 GB/s assumed in our simulation; Table I) moving evicted access metadata to and from memory [31]. Private cache line recalls during LLC evictions in MESI-based CE$^+$ cause CE$^+$ to incur many AIM evictions, and AIM lines are large, especially at 32 cores. Many expensive AIM evictions for canneal lead CE$^+$ to saturate *both* the on-chip interconnect (153 GB/s, greater than the assumed on-chip network bandwidth of 100 GB/s) and the off-chip memory network (180 GB/s). As discussed in Section VI-A, we model network saturation and queuing by scaling execution cycles. Limited bandwidth increases CE$^+$'s execution cycles by 5.8X, and hence CE$^+$ performs poorly compared with CE. As a result, on 32 cores, CE$^+$ is 0.3% slower and uses 0.8% more energy on average than CE.

ARC outperforms CE for several programs and performs similarly for the others. ARC's performance benefit over CE arises from performing fewer memory accesses for metadata. The results also show that AIM and Bloom filter dynamic energy costs are insignificant compared to energy costs of the cache and memory subsystem, justifying their inclusion. ARC performs nearly identically with CE$^+$ for 4–32 cores, and especially outperforms both CE and CE$^+$ for canneal at 8, 16, and 32 cores. ARC's approach to coherence and its use of a non-inclusive LLC stress the network much less for canneal (90 GB/s for on-chip interconnect and 75 GB/s for off-chip memory bandwidth at 32 cores) in moving around metadata compared to CE and CE$^+$ that build on MESI's eager invalidation-based protocol. In general, ARC uses several times less network bandwidth than CE and CE$^+$ for several programs. ARC uses less energy than CE and CE$^+$ for canneal because ARC runs the application faster (i.e., fewer cycles).

For fluidanimate, ARC's execution time is slightly higher than CE$^+$'s (and sometimes CE's, depending on the core count). fluidanimate performs more synchronization operations with increasing numbers of threads, and has progressively smaller regions with more threads (Table II). For ARC, more frequent region boundaries 1) cause more frequent invocations of pre- and post-commit, read validation, and self-invalidation operations, which add execution cycles, and 2) incur latency from cache misses due to frequent self-invalidation.

Sensitivity to AIM size: We evaluated the impact of the AIM size with an *idealized* AIM that has one entry for each LLC line and a smaller-sized AIM (detailed results omitted for space). For simplicity, we evaluated AIM size sensitivity only for ARC. On 32 cores, the idealized AIM improves execution time and energy usage by ~10% compared to the default 64K-entry AIM (Table I). At a lower hardware cost, a 32K-entry AIM increases execution cycles and energy usage by <10% on average, compared to the 64K-entry AIM. These results show that the AIM remains effective at reasonable sizes.

Hardware costs: CE and CE$^+$ differ primarily in the use of an AIM. We estimate the opportunity cost of the AIM in CE$^+$ at 32 cores by translating the space overhead of the AIM into additional LLC size in CE. This configuration of CE, *CE-Ext*, has a a larger LLC (84 MB) than the default at 32 cores (64 MB, see Table I) to account for the AIM overhead. The evaluation methodology is the same as for Figure 7. In

Write buffer size	# Cores 4	8	16	32
8K entries	2.14	3.03	3.88	5.11
16K entries	1.81	2.29	3.06	4.13
32K entries	1.70	2.14	2.79	3.68
64K entries	1.63	2.00	2.53	3.26

TABLE III
Execution time of ARC-TCC for different core counts and per-core write buffer sizes, normalized to ARC for the same core count.

our experiments, the improvement in hit rates due to a larger LLC in CE-Ext is negligible compared to the overall execution, and hence the larger LLC in CE-Ext has negligible impact (<1% on the average) on the overall performance and energy consumption (results omitted for space).

While CE and CE$^+$ build on MESI, ARC avoids a cache coherence protocol. Assuming an idealized sparse directory that distributes state by chaining pointers in cores' private lines, in a system with 8 cores, a 16MB, 16-way LLC with 64-byte lines, a directory would require 1MB of storage for tags and pointers. An AIM for the same system requires ~3MB of storage, adding modest hardware overhead while potentially limiting the need for frequent memory accesses for most applications and different core counts. Furthermore, ARC's use of release consistency and self-invalidation mechanisms provides more design flexibility by not requiring an inclusive LLC and support for core-to-core communication.

Other than the space overhead, detailed results from McPAT show that static power dissipation from the AIM contributes to overall power insignificantly.

C. Comparison with TCC

Transactional Coherence and Consistency (TCC) is a hardware transactional memory (Section VII) design that, like ARC, provides coherence and conflict detection at region boundaries without a MOESI-like protocol [20]. As in CE, CE$^+$, and ARC, all code in TCC executes in regions (i.e., transactions). TCC broadcasts transaction write sets to detect conflicts at region boundaries. Speculation allows TCC to efficiently track accesses for regions of memory larger than a byte (e.g., cache line), but coarse granularity leads to false conflicts. To compare TCC with ARC empirically, we evaluate a modified version of ARC called *ARC-TCC* that uses TCC's mechanisms. For ARC-TCC, we compute execution cycles *excluding* read validation and pre- 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 [20].

Table III shows the run-time overhead incurred by ARC-TCC compared with ARC. The amount of serialization incurred by ARC-TCC or TCC during an ongoing transaction depends on how often the per-core write buffer in the TCC architecture overflows. To estimate the impact of the write buffer on ARC-TCC's performance, we evaluated the performance of ARC-TCC for write buffer sizes of 8–64K per core. For each core count and write buffer size, Table III reports the ratio of ARC-TCC to ARC's execution time. The

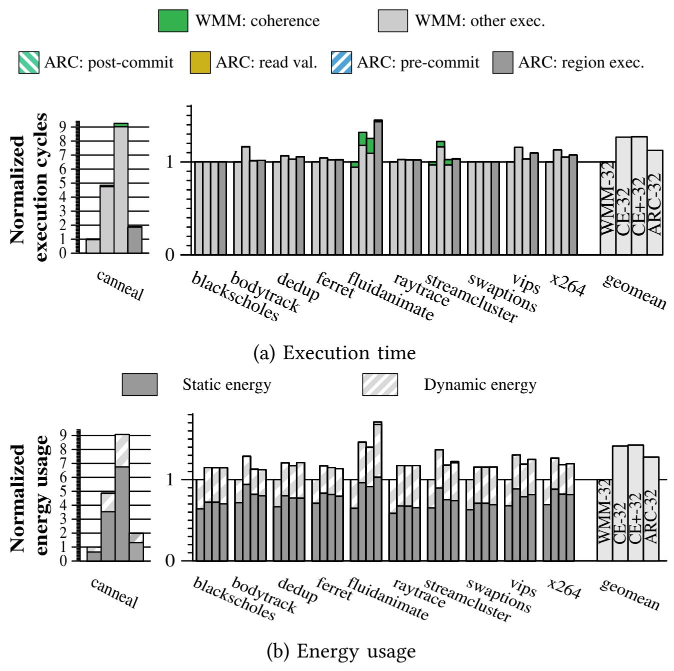


Fig. 8. Execution time and energy usage for CE, CE$^+$, and ARC for 32 cores, normalized to WMM with 32 cores.

table shows that TCC's mechanisms continue to incur high run-time overhead even with large per-core write buffers because many regions overflow the private caches, leading to much serialization. This comparison shows that, for the same context (i.e., precise conflict checking of SFRs), ARC's design provides substantial performance benefits over TCC.

Follow-up work optimizes TCC using a directory [10] and by parallelizing commits [37]. However, TCC is fundamentally limited by its use of *bounded* write buffers, which overflow, leading to serialized commit.

D. Evaluating the Cost of Strong Memory Consistency

Modern shared-memory systems provide *undefined semantics* for programs with data races (Section I). *Memory models* for shared-memory programming languages such as C/C++ and Java are mostly unable to provide useful guarantees to executions with data races [1], [7], [33]. Though hardware memory models (e.g., [3], [46], [50]) are generally stronger than language models, they apply only to the compiled code, thereby failing to provide *end-to-end* guarantees with respect to the source program. Here, we estimate the cost of providing SFRSx over such a *weak memory model* (WMM). Note that *WMM is not directly comparable to this paper's approaches, given the different guarantees provided.*

Our WMM configuration models the same directory-based MESI protocol (Figure 8.6 in [49]) used by CE (Table I). Figure 8 shows how CE, CE$^+$, and ARC compare to WMM for 32 cores, using the same methodology as for Figure 7, normalized to WMM-32.

Figure 8(a) shows that on average, CE, CE$^+$, and ARC are slower than WMM by 26.7%, 27.1%, and 12.5%, respectively, for 32 cores. Figures 7(a) and 8(a) show that CE, CE$^+$, and ARC scale well for most programs, except for canneal and fluidanimate. The energy usage is proportional to the running time of each configuration, and is also influenced by the frequency of accesses to the AIM cache and the Bloom filter structures for relevant configurations other than WMM. Figure 8(b) shows that on average CE, CE$^+$, and ARC use 41.4%, 42.6%, and 27.8% more energy than WMM. CE, CE$^+$, and ARC have particularly high overhead for canneal and fluidanimate. As discussed in Section VI-B, canneal has large regions and working sets, which either saturate the on-chip interconnect and the off-chip network for CE and CE$^+$, or require more memory accesses for ARC to transmit access information compared to WMM, significantly slowing execution and increasing energy usage. fluidanimate has short regions that fail to amortize the cost incurred by the operations at region boundaries for configurations other than WMM.

E. Summary

CE provides well-defined semantics for all executions, but incurs a substantial cost compared to WMM to maintain precise byte-granular access information and to check for region conflicts. CE$^+$, the first contribution in this work, can potentially improve performance and reduce energy usage for a number of applications across core counts, but with increased complexity.

ARC, the second contribution, is a completely different design, which performs well compared with CE and CE$^+$. Our work shows that detecting region conflicts using coherence based on release consistency and self-invalidation can be competitive with techniques that either rely on eager invalidation-based coherence (e.g., CE) or are impeded by fundamental limitations on region size (e.g., TCC). Furthermore, we show that ARC can provide strong consistency guarantees with performance that is competitive with the performance of current shared-memory systems (WMM).

VII. Related Work

This section compares CE$^+$ and ARC with related work *not* already covered in Section II.

Valor provides SFRSx in software alone but slows executions by almost 2X on average [6]. *IFRit* likewise adds high overhead to detect conflicts between extended SFRs [15]. Ouyang et al. *enforce* SFR serializability using a speculation-based approach that relies on extra cores to avoid substantial overhead [36]. *SOFRITAS* enforces software-based conflict serializability through fine-grained two-phase locking [13].

Hardware transactional memory (HTM) also detects region conflicts [21], [23]. However, HTM systems can use *imprecise* conflict detection by leveraging speculative execution, while SFRSx requires precise conflict detection; and HTM must keep original copies of speculatively written data, in case of misspeculation. Like CE and CE$^+$, most HTMs piggyback conflict detection on the cache coherence protocol [35], [54]. Unbounded HTM designs incur run-time cost and design complexity because data that leave the cache cannot easily be tracked by the coherence protocol (e.g., [4]).

BulkSC resembles TCC but broadcasts imprecise *write signatures* [8], [9]. To ensure progress, BulkSC dynamically subdivides regions, precluding its application to SFRSx's unbounded regions.

Software transactional memory (STM) can handle unbounded regions without hardware modifications, but requires heavyweight instrumentation and synchronization that slows (single-thread) execution by 2X or more [12], [21], [22], [41]. Some STM systems use version or value validation of reads (e.g., [12], [21]). ARC's adaptation of validation to the hardware cache hierarchy and its combination of version and value validation are both novel.

DeNovoSync, *SARC*, and *VIPS* use self-invalidation to reduce complexity compared to M(O)ESI [26], [38], [51]. *TSO-CC* and *Racer* provide TSO using self-invalidation and without tracking sharers [17], [40]. *DeNovo* and *DeNovoND* use self-invalidation for coherence, assuming DRF [11], [52]. Jimborean et al. use compiler analysis that assumes DRF to safely extend SFRs, reducing self-invalidation costs [25]. *Distributed shared memory* systems use release consistency to reduce traffic and latency [27]. Unlike prior work, ARC does *not* assume DRF. Instead, ARC exploits synergy between mechanisms for coherence and conflict detection, detecting data races that manifest as SFR conflicts to provide SFRSx.

Prior work supports memory models based on serializability of *bounded* regions that are in general shorter than full SFRs [2], [34], [43], [47]. *Sequential consistency* (SC) is essentially serializability of single-instruction regions [29], [30], [48]. To provide end-to-end guarantees, all of these approaches require corresponding restrictions on compiler optimizations. *DRFx* detects conflicts among bounded regions by maintaining region buffers and Bloom filter signatures of memory accesses [34], [47]. DRFx broadcasts the Bloom filter signatures and occasionally the region buffers across cores, which is unscalable for large regions (e.g., SFRs) and with increasing core counts.

Researchers have introduced custom hardware to accelerate data race detection, extending cache coherence and adding on-chip vector clock metadata [14], [42], [53].

VIII. Conclusion

CE$^+$ and ARC are architecture designs that ensure strong, well-defined semantics for all executions, including executions *with* data races. Compared to the state-of-the-art technique CE [31], we show that an AIM cache in CE$^+$ seems promising to reduce the cost of providing SFRSx. The key to ARC's efficiency is its novel design that builds on and leverages release consistency and self-invalidation mechanisms. ARC outperforms CE and TCC [20], and performs competitively with CE$^+$ in terms of run time and energy usage. These results suggest that CE$^+$ and especially ARC advance the state of the art significantly in parallel architecture support for region conflict exceptions.

Acknowledgments

We thank Joe Devietti for sharing the RADISH simulator; Vignesh Balaji and Nathan Beckmann for technical suggestions; and Nathan Beckmann, Alberto Ros, and the anonymous reviewers for detailed and insightful feedback on the text.

References

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

[2] 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*, 2009, pp. 133–144.

[3] 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*, 2008, pp. 13–24.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *HPCA*, 2005, pp. 316–327.

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

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

[7] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in *PLDI*, 2008, pp. 68–78.

[8] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA*, 2007, pp. 278–289.

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

[10] 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*, 2007, pp. 97–108.

[11] 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*, 2011, pp. 155–166.

[12] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: Streamlining STM by Abolishing Ownership Records," in *PPoPP*, 2010, pp. 67–78.

[13] C. DeLozier, A. Eizenberg, B. Lucia, and J. Devietti, "SOFRITAS: Serializable Ordering-Free Regions for Increasing Thread Atomicity Scalably," in *ASPLOS*, 2018, pp. 286–300.

[14] 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*, 2012, pp. 201–212.

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

[16] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A Race and Transaction-Aware Java Runtime," in *PLDI*, 2007, pp. 245–255.

[17] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *HPCA*, 2014, pp. 165–176.

[18] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective Data-Race Detection for the Kernel," in *OSDI*, 2010, pp. 1–16.

[19] 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*, 1990, pp. 15–26.

[20] 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*, 2004, pp. 102–113.

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

[22] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing Memory Transactions," in *PLDI*, 2006, pp. 14–25.

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

[24] HP Labs, "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," https://www.cs.utah.edu/~rajeev/cacti7/.

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

[26] S. Kaxiras and G. Keramidas, "SARC Coherence: Scaling Directory Cache Coherence in Performance and Power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, 2010.

[27] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *ISCA*, 1992, pp. 13–21.

[28] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO*, 2009, pp. 469–480.

[29] C. Lin, V. Nagarajan, and R. Gupta, "Efficient Sequential Consistency Using Conditional Fences," in *PACT*, 2010, pp. 295–306.

[30] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram, "Efficient Sequential Consistency via Conflict Ordering," in *ASPLOS*, 2012, pp. 273–286.

[31] 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*, 2010, pp. 210–221.

[32] 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*, 2005, pp. 190–200.

[33] J. Manson, W. Pugh, and S. V. Adve, "The Java Memory Model," in *POPL*, 2005, pp. 378–391.

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

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

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

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

[38] A. Ros and S. Kaxiras, "Complexity-Effective Multicore Coherence," in *PACT*, 2012, pp. 241–252.

[39] ——, "Callback: Efficient Synchronization without Invalidation with a Directory Just for Spin-Waiting," in *ISCA*, 2015, pp. 427–438.

[40] ——, "Racer: TSO Consistency via Race Detection," in *MICRO*, 2016, pp. 1–13.

[41] 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*, 2006, pp. 187–197.

[42] C. Segulja and T. S. Abdelrahman, "Clean: A Race Detector with Cleaner Semantics," in *ISCA*, 2015, pp. 401–413.

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

[44] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic Race Detection with LLVM Compiler," in *RV*, 2012, pp. 110–114.

[45] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi, "Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management," in *ISCA*, 2015, pp. 79–91.

[46] 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*, vol. 53, no. 7, pp. 89–97, 2010.

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

[48] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-End Sequential Consistency," in *ISCA*, 2012, pp. 524–535.

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

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

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

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

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

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