Racer: TSO Consistency via Race Detection

Alberto Ros
Department of Computer Engineering
Universidad de Murcia, Spain
aros@ditec.um.es

Stefanos Kaxiras
Department of Information Technology
Uppsala Universitet, Sweden
stefanos.kaxiras@it.uu.se

***Abstract*—Several recent efforts aim to simplify coherence and its associate costs (e.g., directory size, complexity) in multicores. The bulk of these efforts rely on program data-race-free (DRF) semantics to eliminate explicit invalidations and use self-invalidation instead. While such protocols are simple, they require software cooperation. This is acceptable only for (correct) software that abides by the SC-for-DRF semantics defined in many modern programming language standards (e.g., C++11, Java, the latest C standards) but many are unwilling to trust coherence that relies solely on program semantics for its correctness.**

To address this important issue, this work proposes Racer, an efficient self-invalidation/write-through approach that guarantees the memory consistency model of the most common family of processors (TSO-x86), and at the same time maintains the relaxed-ordering advantages of SC-for-DRF protocols.

Lacking a directory and explicit invalidations, Racer achieves this by detecting read-after-write races and causing self-invalidation on the racing reader's cache. Racer also uses a coalescing store buffer (at the L1 level) that allows coalescing and reordering of stores but upon detecting a race, delays the racing read until all its stores appear in order to the read. Race detection is performed using an efficient signature-based mechanism at the level of the shared cache.

Racer performs significantly better than a traditional non-scalable directory-based protocol that does not allow reordering at the protocol level (14.2% in time and 26.4% in energy), a directory protocol for TSO (1.9% in time and 15.5% in energy), and state-of-the-art SC-for-DRF protocol that relies on acquire-release annotations in the programs (6.7% in time and 9.5% in energy). Racer self-invalidates less than program-level annotations as it only enforces ordering on dynamically detected races and provides significant reductions in network traffic and memory-system energy consumption.

I. Introduction

Coherence protocols are designed to hide the effects of caching from the underlying consistency model [1]. Typically a coherence protocol enforces a single-writer-multiple-readers (SWMR) invariant and guarantees propagation of writes so that the effects of caching become transparent to the consistency model, even for sequential consistency (SC) [2].

Recently, however, there has been a slew of work that *breaks the ordering guarantees at the coherence protocol level* in an effort to simplify coherence and its associated costs [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. The bulk of these works focuses on simple protocols without invalidation that rely on data-race-free (DRF) semantics [15]. We collectively refer to them as SC-for-DRF protocols. The main tool in these protocols is self-invalidation (SI) applied on synchronization. Two benefits of SI are the elimination of invalidation messages and the reduction of the directory as sharers do not have to be tracked to be invalidated.

978-1-5090-3508-3/16/$31.00

These protocols require software cooperation. This is straightforward for software that abides by SC-for-DRF semantics defined in many modern programming language standards (e.g., C++11, Java, the latest C standards [16], [17], [18]). DRF semantics mandate that all synchronization be visible to the hardware [15].

While some SC-for-DRF proposals advocate use in relation to "disciplined programming" [3], others propose a low-level approach [5]. However, designers and many researchers are unwilling to consider a protocol that relies solely on program semantics for correctness. Critical questions emerge: What about legacy code (or legacy libraries)? What if someone forgets to annotate (and expose) a synchronization? What if a program steps outside language standards? Programs *do* have data races [19], [20]. While SC-for-DRF protocols are appealing because of their simplicity and reduced cost, the question is how can one achieve their benefits *without* requiring the cooperation of software for correctness.

In this paper, we propose a simple self-invalidation, self-downgrade protocol, called *Racer*, that does not require any involvement of the software. The key intuition driving our work is that given any program, *if we detect its races at runtime and dynamically call the racing reads "acquire synchronization"* we can enforce memory ordering through an *implicit* self-invalidation fence in the racing reader, exactly as an SC-for-DRF protocol would do with explicit knowledge of the synchronization. Similarly, in the presence of a race, we ensure that stores are observed in a valid order. Because Racer uses store buffers and self-invalidation caches it violates SC and results in a consistency model known as total-store order (TSO) [21]. There are four key benefits with this approach:

Less synchronization: In SC-for-DRF *every* conflicting access must be separated by synchronization. Thus, acquire synchronization in a DRF program, is possibly an overkill as the program must fend against any possible race in execution. As shown by von Praun et al., many of the possible races never actually materialize in a given execution [22]: Java programs with acquire-release style synchronization (locking) contain many superfluous memory fences (for races that do not actually appear) forcing unnecessary memory ordering. Thus, acquire synchronization detected at runtime is potentially *less*

than the (acquire) synchronization that a DRF program would employ. This is a powerful property, captured in our approach, as it enforces no more than the necessary ordering for *a* specific execution.

Relaxed load→load order in the absence of races: Relaxing load→load order is particularly important for performance. Without this relaxation, non-blocking caches, multiple outstanding misses, and even memory-level parallelism (MLP) would not be effective. Load→load order can be relaxed for both SC [23] or TSO [24] *as long as this relaxation is not "seen" by races.* Racer detects data races during execution and easily manages load→load relaxations.

Relaxed store→store order and store-wait-free operation: While a store buffer (TSO) is essential for performance, many workloads experience significant blocking due to its limited size and the need to enforce store→store order (required in TSO). An extended coalescing store buffer at the L1 [24], solves the problem by relaxing store→store order but relies heavily on expensive speculation, checkpointing (outside the core), and rollback in the case of a race. Racer delays the response to a racing read and offers the same extended L1 store coalescing capabilities but without any speculation, checkpointing or rollback cost, effectively providing store-wait-free operation by default.

Implementation Efficiency: To detect runtime races we are inspired by efficient techniques that were developed for thread-level-speculation (TLS) and transactional memory (TM) (e.g., Ceze's et al. "Bulk" [25]). While such methods were developed *given* an underlying SC-capable protocol (based on invalidation), we turn the table and now use race-detection not to detect memory dependence violations due to speculation (TLS) or unsynchronized execution (TM) but in fact to enforce consistency with self-invalidation and *buffered* write-through (also known as self-downgrade [12]). We use signatures (one per core) to record and detect unseen writes.

Resulting memory model: Self-invalidation and write-through buffers violate a required ordering for SC (store→load) and therefore, with such a starting point we cannot offer SC (e.g., correctly run Dekker's algorithm—Section V). However, few contemporary systems implement SC (the IBM Series Z comes close [26]). The bulk of the systems in use today implement TSO or weaker memory models. Our approach provides TSO ordering (Section V) and thus is compatible with the vast majority of systems and software.

Contributions: In this paper, we propose *Racer*:

- A novel self-invalidation/write-through approach that provides TSO and: i) requires *no support from the software nor any DRF guarantees*; ii) eliminates directory, explicit invalidations, tracking of the sharers/writers, and indirection; iii) does not rely on timestamps (to derive ordering) and their associated overheads; and iv) does not impose a coarse coherence-granularity that can result in false-sharing. Since Racer is strictly request/response it allows for virtual caches as shown in [6]. (Section III)
- Efficient signature-based race detection to drive self-invalidation and write-through. (Section III-C)
- Self-invalidation on run-time detected races, enforcing only the necessary memory ordering for an execution.
- Simple and efficient L1 coalescing store buffers, integral to Racer, that achieve the same benefits as prior, more complex, store-wait-free proposals based on speculation, checkpointing, and rollbacks [24]. Racer coalescing store buffers are non-speculative, allow unordered cache-line write-throughs, while still preserving the observed TSO store→store orderings. (Section III-E)
- Simple prediction (based on race detection) of program acquire/release points for fast propagation of release-stores and fast detection of future races. (Section IV)
- Straightforward scaling to distributed last-level cache (LLC) banks. (Section III-G)

We evaluate Racer using a wide range of Splash2 and Parsec benchmarks and find that it improves by 6.7% the performance of an efficient SC-for-DRF protocol (modeled after VIPS-M [5]) which requires static annotations for the synchronization in the programs. At the same time, it reduces the amount of self-invalidated data by 51.0%, resulting in energy savings of 9.5% in the TLB, caches, and network. Compared to a non-scalable directory protocol it improves execution time by 14.2% and energy consumption by 26.4%. All this is achieved using only very modest race detectors for a total storage lower than 18 KiB per core.

II. Background and Motivation

A large body of work aims to optimize coherence. Comparing various proposals is difficult, since there are many factors that should be considered. Some proposals optimize an aspect of a protocol but often have to compromise on others. Table I gives an overview (for a number of interesting recent proposals) focusing on some common weaknesses in protocol designs:

- **DRF:** The protocol requires DRF semantics. While DRF semantics simplify coherence [3], [5], [7], [12], [13], [27], the resulting proposals cannot support legacy code that is not guaranteed to adhere to SC-for-DRF. One exception is RC3 [14] that although not tied to DRF semantics, requires them for efficiency reasons.
- **Dir:** The protocol is based on an invalidation directory. A large body of work aims to offer scalable invalidation directories [28], [29], [30], [31], [32], but often at the cost of increasing complexity.
- **Timestamps:** The protocol is based on timestamps. Such protocols [8], [14], [33], [34], [35] effectively derive ordering information via global clocks or local optimized Lamport-like clocks [36] to enforce strong memory models (SC or TSO). Timestamp overhead and complexity is comparable to that of directory protocols (albeit more scalable). Supporting SC is likely not required in practice as few contemporary systems support it (e.g., IBM Series Z). The

TABLE I
PROTOCOL WEAKNESSES

		Area & Complexity				Protocol		
Protocol	Strength	DRF	Dir	Time	Bcast	False-S	Ind/Fwd	Slow-Rel
Dir_N	SC	—	✓	—	—	✓	✓	—
Snooping	SC	—	—	—	✓	✓	✓	—
SPEL [37], [38]	SC	—	✓	—	—	✓(*)	✓	—
TARDIS [34]	SC	—	—	✓	—	✓	✓	✓(*)
TSO-CC [33]	TSO	—	—	✓	✓(*)	✓	✓	✓(*)
RC3 [14]	TSO	✓(*)	—	✓	✓(*)	✓	✓	✓(*)
TC-weak [8]	SC-for-DRF	✓	—	✓	—	✓	✓	✓(*)
DeNovo/ND [3], [7]	SC-for-DRF	✓	—	—	—	✓(*)	✓	—
DeNovoSync [13]	SC-for-DRF	✓	—	—	—	✓(*)	✓	✓(*)
Dir_1-SISD [27]	SC-for-DRF	✓	—	—	—	—	—	✓(*)
VIPS [5]	SC-for-DRF	✓	—	—	—	—	—	✓(*)
Callbacks [12]	SC-for-DRF	✓	—	—	—	—	—	—
Racer	TSO	—	—	—	✓(*)	—	—	—

✓(*) indicates only in some cases

protocols that support TSO (TSO-CC [33], RC3 [14]) relax timestamp-derived ordering to fit a less strong consistency model than SC, but still suffer the full overheads of timestamps (same as their SC-capable counterparts).

- **Broadcast:** The protocol requires Broadcast. Broadcast generates traffic, limits scalability, and requires network support. Broadcast on corner cases, e.g., read-only optimization [14], [33], incurs the full cost and complexity of supporting broadcast even if it is *rarely used*.
- **False-Sharing:** Protocols that keep coherence information at a cache-line granularity suffer from false-sharing that is detrimental to performance. SPEL [37], [38] avoids false sharing only for DRF accesses, while DeNovo [3], [7], [13] employs dirty and touched bits at the designated DRF granularity.
- **Indirection and forwarding:** Indirection (via the directory or the LLC) increases complexity, slows resolution of reads, and complicates virtual-cache coherence [6]. Only strictly request-response protocols avoid indirection completely.
- **Slow Release:** Protocols that use timestamps and implement leases or expirations for cache lines, can delay write propagation. Other approaches avoid premature invalidations by using exponential back-off [5], [13], but at the cost of not seeing writes immediately.

While our analysis is by no means complete with respect to all the weaknesses that one can examine in a coherence protocol, we note that for those we examined, no protocol seems to be "weakness-free" (Table I). Many works target a set of properties but in the process they have sacrifice (to a greater or lesser degree) some others: even Callbacks [12], which seems to eliminate many of the weaknesses, relies *heavily* on DRF and in fact imposes *increased* burden to the programmer beyond simply annotating synchronization. The proposal in this paper, *Racer*, eliminates most of these weaknesses.[1] Compared to the state-of-the-art, performs equally well, while leading by a wide margin on traffic- and energy-reduction.

[1]Only requiring broadcast in some cases which, however, can efficiently implemented with global wires.

III. ENFORCING CONSISTENCY WITH RACE DETECTION

An SC-for-DRF protocol provides coherence by self-invalidating and writing-through shared data on synchronization points. These operations take the form of fences (e.g., as in [12]): synchronization—which actually constitutes races—is exposed to the protocol by fences inserted in the code.

Our challenge is to invoke self-invalidation/write-through fences without any a-priori knowledge of program synchronization. Fundamentally, we seek to detect races during program *execution* which we can equate to synchronization points. We assign acquire semantics on a read involved in a race (causing an implicit self-invalidation fence on the reading core) and release semantics on a racing write (ensuring that the store order of the writer is correctly observed by the racing read). All other accesses not involved in a race effectively become data-race-free. These accesses are unordered between races but become ordered with respect to the races. The resulting memory model as we will show (Section V) is TSO.

A. Private/Shared Data Classification

We start with a self-invalidation/write-through protocol because of its simplicity: lack of directory to track sharers and writers, lack of explicit invalidations, lack of indirection, and lack of timestamps (to detect and enforce ordering). We assume that we have private caches that are self-invalidated on demand (SI-caches). The first step to make self-invalidation effective is to restrict it to "shared" data. Data classification into Private and Shared is widespread [30], [39], [40], [41], [42], [43] and can be done with either a page-base OS-assisted approach [5], [30], [39], [44] or with hardware solutions [27], [32], [43], [45]. The actual approach is not relevant to this work as it is an orthogonal optimization; we use an OS-assisted approach for the evaluation, but equally well could chose any other.

We self-invalidate only the shared data in the local SI-caches. Private data do not affect consistency [42]. Thus, accesses to private data do not invoke any of the mechanisms described herein, relieving such mechanisms from significant pressure. The only requirement to satisfy is that in a private to shared transition, the corresponding dirty data must be written through to the LLC before allowing the access that causes the transition to continue. This is standard practice in all previous works.

B. Overview

Central to Racer is a structure called RAWR that detects Read-After-Write races. This structure is a signature-based table that sits at the level of the shared LLC cache[2] (see Section III-C).

A load that *misses* in the L1 sends a request to the LLC where the RAWR is located. If a race is detected in the RAWR then the following actions are taken:

1) The LLC access is delayed until possibly outstanding stores from the writer core are made globally visible in

[2]For simplicity we assume a 2-level cache hierarchy where L1 represents collectively a core's private caches and LLC represents the shared cache.

a correct order. Two other small helper structures, the Outstanding-Store Owner (OSO) and the Ack counter Array are used for this purpose (see Section III-E).

2) After the LLC access, the response to the core enforces an implicit self-invalidation fence (SI-fence) before the load is serviced with the requested data.
3) The SI-fence waits for all outstanding loads prior to the racy load to complete, squashes speculative loads that issued after the racy load, and then causes a self-invalidation of all the *shared* data in the core's L1.

When do we check for races? Not every load is sent to the RAWR—only L1 misses go to the RAWR as they have to go to the LLC anyway. In a self-invalidation cache, since we have no explicit invalidations, this would allow a load to hit indefinitely on stale data, instead of detecting a race.

To prevent this from happening, each L1 cache line can only be accessed for a limited time before causing a (non-blocking) *check* for a race. A small decay-like coarse-grained counter per cache-line (e.g., 2-bit), ticks every 1000 cycles. When the counter saturates, the next access to the cache line emits a check and resets the counter.

The check for a race only invalidates the corresponding L1 cache line *if it detects a race* in the RAWR —no further action is taken. This solitary invalidation, causes the *next* access to miss, detect the race, and self-invalidate all the shared data in the L1 via an implicit SI-fence. This simple approach works well in the common case of spin-loops, while causing minimal disturbance otherwise. In contrast, another similar approach, TSO-CC, allows a cache line to be accessed for a limited number of times before it *invalidates and causes a miss* [33], [14]. In TSO-CC invalidation of the cache line is unavoidable (does not depend on race detection) and the cache line counter must be ticked with every access to the cache line.

Granularity: An issue in many cache coherence protocols is that they maintain coherence information (e.g, in directories) on a cache-line granularity. This can lead to disruptive *false sharing*. In Racer, changing the granularity from cache line to word (or byte) is straightforward because it is based on signatures and the cost remains the same. However, additional support is required in the caches (per-word dirty bits) and some operations become more complex (e.g., reading a cache line from the LLC requires checking for races on every word). Since the impact of false sharing on our benchmarks is not great we consider only cache line granularity for the rest of the paper.

C. The RAW race detector

At the heart of Racer is a structure, called RAWR (RAW-Race detector), that records, in a concise manner, stores that a core has not "seen" yet. RAWR uses an array of signatures (Bloom filters [46]), one filter per core (Figure 1). [3]

[3]Signatures in this figure are simple one-bit address hashes (for illustration purposes) but in reality we have tested the approach with different Bloom filter implementations including H3 [47] (both with single and multiple hash functions), Bulk [25], and Xor (LogTM [48]). For the evaluation we employ Bulk signatures as they provide the best performance.

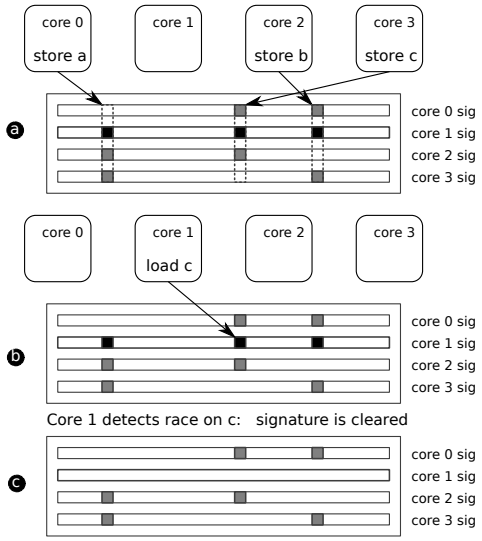

Fig. 1. An example of the operation of the RAWR.

To explain the RAWR operation we will assume for the moment that each store (target address and data) is sent to the LLC in order. A store inserts its target address in the signature of all *other* cores —not of its own core— and stores its data in the LLC. In Figure 1.a for example, core 0 inserts address `a` into the signatures of cores 1,2, and 3. A core has not "seen" any of the stores in its RAWR signature until it tries to read from an address that is already inserted there by a store (e.g., core 1 accessing address `c` in Figure 1.b). At that moment a RAW race is detected.

When a core detects a race in RAWR, we clear its signature (Figure 1.c), and begin recording new stores in it. Clearing the signature is a simple set-row-to-zero operation, and it is done because it gives more accurate race detection—not for correctness.

Self-cleaning: A useful property of the RAWR is that it is self-cleaning when races appear infrequently. With the passage of time, signatures become saturated with stores, raising the possibility of detecting spurious races due to aliasing. When this happens signatures are cleared, allowing for a new set of stores to be recorded. A spurious race causes an extra self-invalidation but this is not a concern if it happens rarely. This is evidenced in our evaluation.

RAWR vs. TagLess: The RAWR structure bears similarity to the *TagLess* directory [28], which also has a signature per core and records *sharers* per address (column-wise) with the intent to invalidate them on a write. The RAWR on the other hand, although similar in structure, is used quite differently. The novelty of the RAWR is not the detection of RAW races per se, as this is straightforward. Rather, it is the way this race detection drives self-invalidation in the core caches and manages write-through to provide TSO. The clearing of the signatures in RAWR is also much simpler (set to zero) than in TagLess directories which need either counter-based Bloom filters or partitioning of the Bloom filters to avoid conflicts [28].

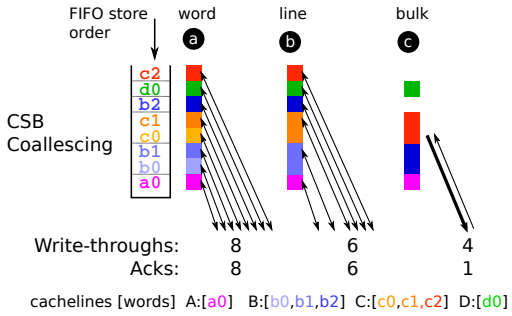

Fig. 2. CSB: (a) word, (b) line, (c) bulk.

D. Collapsed Order

A core cannot discern order in the stores it sees upon detecting a race. It appears to a core that all the stores (since the core's last race detection) were performed together. Each core has a different picture of the set of stores that appear to be performed together, depending on the detection of races on that specific core. For this reason, we refrain from saying that all stores seen by a core have been performed "atomically." Instead, we refer to this as *Collapsed Order* meaning that a core discerns only one order (all stores at the same time), but that does not necessarily mean that the same stores are unordered for other cores.

As Wenisch et al. have shown [24], and as we will explain in detail below for our approach (Section V), seeing at once all the stores that precede a racy store is sufficient to support the observable TSO orderings.

E. Store coalescing

In Collapsed Order all the stores preceding a racy store must appear at once to the core that detects the race. This allows a general coalescing store buffer, but additional guarantees must be given in the presence of a race.

We assume an L1-level store buffer, called Coalescing Store Buffer (CSB), as shown in Figure 2. Conceptually, this store buffer could be integrated with the core's store buffer but in this paper, we describe it independently.

The core emits loads, stores, atomics and fences, to the L1. A store writes its data in the L1 and enters its target address in the CSB. Memory fences and atomic instructions (i.e., atomic RMW) drain both the core's store buffer and the CSB before executing (atomics) or committing (fences). Evictions of L1 lines that are resident in the CSB require draining of the CSB (explained below). Lastly, a timer causes the draining of the CSB (and the core's store buffer) after some period of write-through inactivity to guarantee forward progress. Similarly to the cache-line timer for the race checks, the CSB timer is a coarse grain counter that starts with the insertion of the oldest entry in the CSB and when it reaches saturation causes a write-through and resets.

We examine three ways that the CSB can be drained (written-through to the LLC):

Word: (Figure 2.a). Store→store order can be naively ensured by writing through each store to the LLC *in order*. The main problem of this option is the serialization of all stores and the amount of traffic generated (for every store, the data message and an ack to confirm that it arrived to the LLC), even if this only concerns stores on shared data [5].

Line: (Figure 2.b). Coalescing *consecutive* stores to the same cache line (e.g., `b0` with `b1` and `c0` with `c1` in 2.b) can still guarantee FIFO order because when the cache line exits the CSB it is written through to the LLC atomically. The CSB keeps the necessary per-word dirty bits to write-through the relevant cache-lines as *diffs* to the LLC, so that all the *modified* words of a cache line appear indivisibly—in Collapsed Order. While better than the first, this option serializes cache-line write-throughs and still requires an ack per cache line.

The above two options expose stores to the LLC (and the RAWR) in order, avoiding the troublesome case of coalescing *across* another cache line: For example, in Figure 2 store `b2` can coalesce with `[b0,b1]` but only across cache line `C`; similarly store `c2` coalesces with stores `[c0,c1]` but only over cache lines `D` and `B`. Exposing cache line `B` could violate store→store order unless `C` and `D` are also exposed at the same time.

Bulk: (Figure 2.c). Bulk write-through relaxes store→store order as long as all stores appear in Collapsed Order with respect to any core that races with any of them. Bulk write-through drains the CSB completely by writing-through *all* its cache lines. The benefit is twofold: it allows any coalescing across cache lines (Figure 2.c) and lets individual cache-line write-throughs proceed in any order to the LLC. Bulk write-through starts either when the CSB is full or when the maximum write-through delay has been reached (for the oldest entry in the CSB). The corresponding modified cache lines are written to the LLC as diffs. While in Bulk write-through, the CSB does not accept any new stores from the core until it empties. Obviously, a double-buffering optimization can be used in this case, where one part of the CSB is draining while another accepts new stores. Only one ack is required for all the contents of the CSB, regardless of the order in which they are written in the LLC (Section III-F).

F. Waiting for a write-through to complete

In Bulk write-through, stores go to the LLC in any order. In TSO, this is correct as long as no other core can observe the reordering (e.g., via a race). But what happens if a race appears in the middle of a Bulk write-through? The key for correct operation is to guarantee that at the very least all stores *preceding in program order* the racy store appear in collapsed order to the racy read.

Racer vs. Store-Wait-Free: The same problem appears in the work of Wenisch et al. for store-wait-free operation [24]. In their approach the "Bulk" write-through is transactional: it is only committed in the absence of a race. If a race intervenes during the write-through, the solution is to abort and revert back to a checkpoint. This kind of speculation, checkpoint and rollback, *outside the core* is very expensive to implement.

Our approach is unique. Instead of relying on speculation and rollback, we simply *delay the racing read until after the*

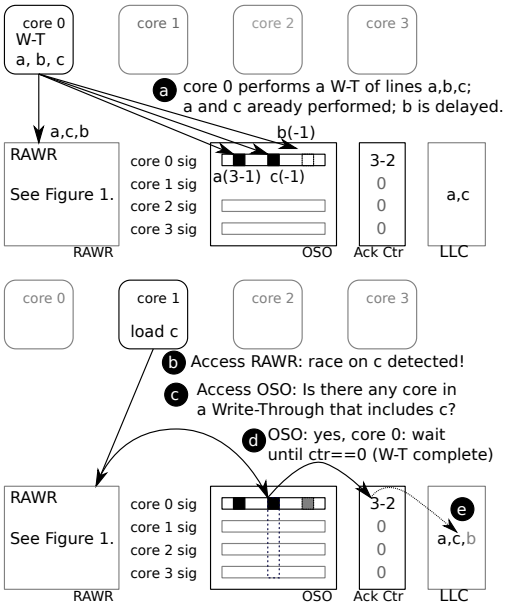

Fig. 3. WT completion upon detecting a race. A core detecting a race in RAWR waits on outstanding write throughs.

write-through completes. To do this we need to know: i) if a racy read races with a store in an ongoing Bulk write-through; and ii) how long to wait until the write-through completes. These two functions are handled by two helper structures, called Outstanding Store Owner (OSO) and Ack counter array (AckCtr), shown in Figure 3 and explained below.

Outstanding-Store Owner (OSO): This structure ensures that races with a store that is part of a Bulk write-through are delayed until the write-through completes in the LLC. The key information supplied by OSO is the "owner" of a racy store.

The OSO is an array of signatures, one per core. The signatures encode the target addresses of a Bulk write-through. In other words, in contrast to the RAWR, a store's target address appears only in the row corresponding to its "owner" core (Figure 3.a). Because the OSO needs to track only the target addresses of a few out-of-order stores, its signatures are much smaller than RAWR's.

Write-throughs insert their target addresses in both the OSO and the RAWR at the same time. The order in which write-throughs update the OSO does not need to respect store→store order.

When a read reaches the LLC it first checks the RAWR for a race (Figure 3.b). If there is a race, the OSO is accessed to see if the race is part of a Bulk write-through from any core. All the rows of the OSO are accessed for a match (Figure 3.c). If there is no match, the read continues to the LLC. But if one is found, the read must *wait* for the "owner" of the Bulk write-through to complete it. This is accomplished by sending the read to the AckCtr array. There, it waits the owner's AckCtr to signal that the write-through has completed (Figure 3.d).

Ack Counter array: The purpose of the AckCtr array is to determine when a Bulk write-trough has been performed: i.e., when all the cache lines are in the LLC. Bulk requires only one ack, regardless of how many cache lines are written-through to the LLC. This is done with a *per core* counter "AckCtr" (Figure 3). Each cache line that is written to the LLC adds a number to the AckCtr. If there are N cache lines to be written, the first one that leaves the CSB adds $N-1$: $AckCtr+=(N-1)$. The rest $N-1$ cache lines, add -1 (each) to the counter. Figure 3.a shows, next to the cache lines a,b, c, the numbers that are added to core0 AckCtr: (3-1) for a, (-1) for b, and (-1) for c. No mater which order, when all the cache lines are written to the LLC, the counter goes to 0. At this point a single ack is returned to the CSB, and any read requests waiting on this counter are released to go to the LLC (Figure 3.e).

Deadlock: The design of Racer guarantees that there are no resource cycles that can deadlock the system. In particular, the draining (write-through) of a CSB cannot be aborted or stalled by actions of another core and it is always guaranteed to complete. In fact, we make sure that CSBs drain even without any activity by using the timer mechanism. Delaying loads in the AckCtr array (until write-throughs complete) also does not create any cycle in the system. A load can only be delayed by a *committed store* of another core, thus no cycle is formed between cores. Finally, limited buffering space in an AckCtr array for queuing reads is also not an issue: when buffering space is exhausted, loads are NACKed back to their MSHR. Thus, in contrast to other approaches (e.g., Wenisch et al. [24]), we are not plagued by deadlock conditions that require additional complexity and effort to resolve.

Evictions: Eviction of dirty cache lines (that have not been written-through yet) drain the CSB: for the Word and Line options they drain in order all the stores that precede them in the CSB; for Bulk write-through an eviction of any cache line in the CSB drains the whole CSB.

G. Multi-banked LLC

The description so far focuses on monolithic RAWR and OSO structures. However, many multicores are tiled (the LLC is distributed in banks to the tiles) or have a NUCA architecture (e.g., such as the recent Sparc M7[49]). In such cases, we assume that the RAWR and OSO are also banked and distributed along with the LLC. Each RAWR and OSO bank is responsible for the blocks that map to their corresponding (local) LLC bank. The AckCtr array is replicated in all banks, i.e., each OSO bank has its own AckCtr array.

In this scenario we need to guarantee that:

- **distributed RAWR:** When a race is detected the clearing of the core's signature (distributed in RAWR banks) happens globally, before the response to the racy access is returned. The bank where the race is detected sends a message to all other banks to clear their part of the signature and waits for confirmation before replying to the racy access. No coordination is needed with respect to stores that can modify the signature in other banks (see the discussion in Section V).
- **distributed OSO:** A *Distributed Bulk write-through* must complete in all banks before allowing the detection of a

race to proceed. This is enforced by extending the Ack protocol (Section III-E) into a two-phase protocol. As in (Figure 3) a racy read that hits in an OSO bank is delayed until the bank's AckCtr reaches 0. However, write-throughs can go to different banks which means that no single AckCtr sums up to 0. To resolve this problem, all the banks send their acks to a single bank and wait a confirmation for the global completion of the write-through. The bank selected to collect the acks is determined by the first cache-line that is written-through by the CSB. The AckCtr in this bank reaches 0 when all the write-throughs complete. At that point it will notify all other waiting OSO banks to erase their signature and zero their AckCtr, thereby releasing also the racy read to proceed to the LLC.

While the above distributed RAWR and OSO protocols imply broadcasts, we note that the messages exchanged in these protocols carry very little information. In fact, they carry only the *function* and the *core ID* for which the function is to be performed. There is no memory address nor any other kind of information and are only *four* functions: `Clear-Signature(core)` and its `ack(core)` for the RAWR protocol and `WT-ack(core)` and `Bulk-complete(core)` for the OSO protocol. This means that the distributed RAWR and OSO protocols can be easily implemented using a number of *Global lines* based on the work of Oh et al. [50], Beckmann and Wood [51], Chang et al. [52] and Krishna et al. [53], instead of injecting messages in the on-chip network. At most $(4 \times \#cores)$ global lines are needed, however a much smaller number of multiplexed lines can be used in large multicores.

IV. Expediting race detection with prediction

Without a priori knowledge of synchronization in a program, race detection can be delayed in two ways: i) stores can be delayed in the CSB and ii) loads can be prevented form reaching the RAWR by not self-invalidating. Similar delays for loads and stores exist in other approaches, for example when "leases" [34] or "expirations" [14], [8] are employed in timestamp protocols. The benefit of Racer is that it detects races and can build on its own knowledge of the program's synchronization.

Expediting acquires with prediction. The policy of limiting the hits to a cache line [33], [14] is straightforward, but—in our experience—can lead to excessive traffic for common idioms such as waiting on barriers. A typical solution would be to implement back-off in software, but that again sets (soft) requirements to software. Our contribution here is to optimize this situation with *instruction-based* prediction. Detecting a load that races (unless it is a spurious race) is a strong indication that we have discovered an acquire synchronization point. We mark this in a small table (e.g., 16 entries for the evaluation) after a few successful race detections (confidence). The next time this load accesses the L1, it immediately invalidates the accessed cache line and proceeds to the LLC and the RAWR. Acquire prediction is a *hint* and does not affect correctness.

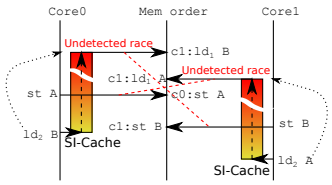

Fig. 4. Dekker's algorithm cannot be run with an SI-Cache (and without detecting WAR and WAW races)

Expediting releases with prediction. Symmetrically to predicting which loads are acquires, we predict which stores are releases. The goal is to drain the CSB and expedite making the buffered stores visible to the other cores, similarly to QuickRelease, proposed by Hechtman et al. but for *explicit* releases in GPUs [9]. To predict a release we rely on the prediction of acquires. A store, following a predicted acquire, that matches the latest target address of the predicted acquire, is marked as a predicted release. We use a small buffer (e.g., four entries in the evaluation) for address matching between the last few predicted acquires and ensuing stores, and keep the resulting release predictions in separate small table (16 entries). Again, release prediction is a *hint* and does not affect correctness.

V. Racer and Consistency Models

SC. SC requires load→load, load→store, store→load, and store→store orders. Both the store-buffer and the SI-cache reorder loads and violate store→load required by SC. The store-buffer is well known to violate this order by delaying stores but letting loads proceed. The SI-cache violates this order by supplying loads with values existing prior to the latest memory write (explained below and in Figure 5).

Consider for example Dekker's algorithm shown in Figure 4. Assume that we have no store-buffer. Just the existence of the SI-cache is enough to violate the store→load. In both cores, loads bypass stores because they hit on stale copies that were loaded prior to the stores.

TSO. TSO relaxes store→load because of the store buffer. The two main challenges that we address in this paper is how to efficiently maintain load→load and store→store orders in an *incoherent* system without core-to-core communication (strict request-response to the LLC). While TSO does not permit load→load and store→store reorderings, these matter only if the reordering can be observed by another core [33], [23], [24]—as we show, this means that a race is involved.

Store→load reorderings. In the example of Figure 5, `ld A` bypasses `st B` in two different ways. Both are legal in TSO as the store→load order is not enforced. (Note that if `A == B`, then the correct program order is ensured in both cases.) In (a) the store is delayed by a write buffer and the load appears *earlier* even if in program order it trails the store. (The delay of the store also causes `core1:ld` not to see the new value.) In (b) the `core1:ld` hits a cached copy (that is not invalidated by `core0:st A`) and therefore it *appears* before `core1:st`. Concerning reordering, the effects of a

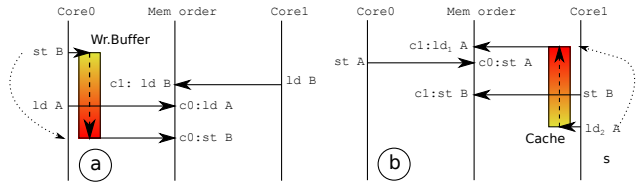

Fig. 5. Store→load reorderings with a store buffer or an SI-cache

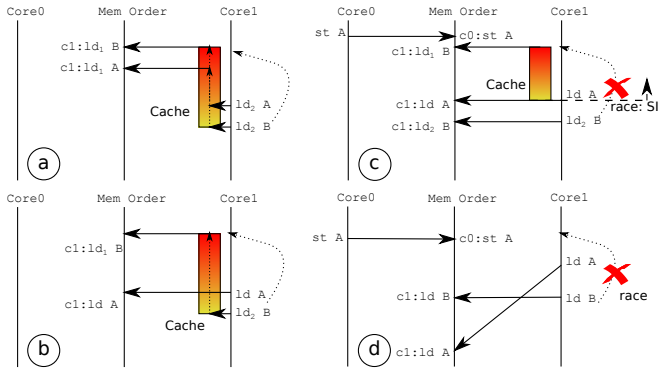

Fig. 6. Load→load reorderings with an SI-cache. A `ld` bypasses a `ld` by hitting in the cache (a and b). The reordering, however, does not matter as, regardless of the order, each load would read the same value (version). A race involving `ld A` prevents the reordering by either self-invalidating the cache (c), or by squashing speculative execution that allows `ld B` to overtake `ld A` (d).

write-buffer and a non-invalidated-cache are indistinguishable. The effect is that loads move "up" and stores "down." As a result, the combination of a write-buffer and a non-invalidated-cache *cannot* violate the load→store order but can only make the load and the store to happen further away.

Racer Load→load reorderings. Load→load reordering occurs when loads hit cached data as shown in Figure 6. The same benign reordering is also implied in TSO-CC [33].[4]. In our case, we specifically enforce order only upon detecting a race.

The reasoning is the following. Assume two loads shown in Figure 6. There are three cases that allow reordering: 1) both loads hit in the cache (Figure 6.a and Figure 6.b), 2) only the second hits (Figure 6.c), 3) both miss but get reordered in the network (Figure 6.d). First, if both loads hit the cache (Figure 6.a) the reordering is benign, since in any order they would obtain the same value for A and B respectively (more precisely the same version of A and B). The same is true if `ld A` misses and goes to the LLC. If there is no race involved then this load "sees" no new value (version), therefore this reordering is also benign. However, if one of them races, and in particular the earlier load (`ld A`) that is overtaken by the later load (`ld B`), the SI caused by its race will prevent the reordering: the race will force the latter load to miss and access the LLC.

ILP load→load reorderings. More broadly, the issue is how Racer treats load→load reorderings that are sourced in out-of-order execution in ILP cores. In the case shown in Figure 6.d both loads miss and get reordered (e.g., in the network). However, this reordering must have its source in ILP, as loads "*block*" for their data response. In other words, `ld B` must be speculative and out-of-order with respect to `ld A` and cannot commit unless `ld A` receives its response and commits. Since we detect a race on `ld A` this means that the value that `ld B` read, even from the LLC, may be incorrect and `ld B` must be squashed and reissued.

[4]E.g., a cached block can be used for a number of times before being self-invalidated to check for new values, at which point load→load order is enforced

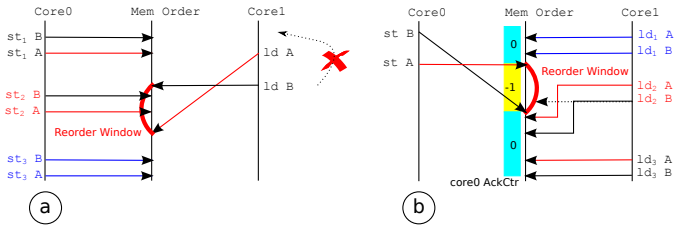

Fig. 7. ILP reorderings

The key here is that as long as the reordering of `ld B` with respect to `ld A` cannot be observed, it is allowed—see an extensive discussion by Gniady et al. [23] and examples of real implementations [54]. In our case, the reordering can only be observed with a pair of stores writing new values on the addresses of the reordered loads. Figure 7.a shows possible cases of stores interacting with a reordered load→load: Clearly if both stores happen *after* the reorder window (`st3`) reordering cannot be observed as the loads do not see any new values. If both stores happen *before* the reordering, the loads see their new values regardless of order and the reordering cannot be observed. In this case it would be `ld B` that detects the race, self-invalidates and lets `ld A` see its new value. The critical case is when `st A` happens *during* the reorder window. There is no guarantee where `ld B` will fall, if it will see its new value, or even if it sees a correct value (i.e., the last before `st A`). Thus, when `ld A` detects a race, it squashes the execution `ld B` that followed but *possibly incorrectly* performed earlier. **The invariant in our approach is that a load is not allowed to bypass an earlier load that races.**

Figure 7.b shows the duality with store→store reorderings discussed in Section III-E. Again, store reordering can be observed by a pair of loads only if the racing load (`ld A`) happens in the reorder window. The difference between load-load and store-store reorderings is in the way we manage the uncovering of the reordering. We do not stop stores from falling in a load-load reorder window, therefore we must squash the out-of-order load when this happens (the ILP core provides this mechanism for free). In contrast, we stall loads in the LLC when they fall in a store-store reorder window. This asymmetry (only loads can stall but never the stores) is what guarantees absence of deadlock in Racer.

Store→store reorderings. Store→store reordering in TSO was shown by Wenisch et al. [24]. They show significant benefits from relaxing local store order in coarser regions

of stores that appear atomic. Their mechanism referred to as Atomic Sequence Ordering (ASO), dynamically creates atomic sequences of stores delineated by the appearance of fences (for TSO) and RMW atomics in the instruction stream. However, races are handled as *disruptions* in the atomic commit of a store sequence, causing an abort. Unfortunately, this demands rollback and replay and of course the all the cost of the corresponding checkpointing mechanisms, including processor state, registers, TLB state, and speculative state in the L1 cache [24].

The fundamental improvement that we provide over the work of Wenisch et al. is that we provide similar store ordering flexibility but with much less complexity and *without speculation*. The key difference is in the way we treat races: instead of aborting an atomic store sequence on a race, we stall the racing read on the core's AckCtr until (at least) earlier stores (to the store participating in the race) appear performed in Collapsed Order to the racing core. In addition, stores in our case never cause write-misses as they never lack permission to write in the L1. Without needing to resort to speculation and rollback we thus achieve the benefits of the Scalable Store Buffer [24] which maintains speculative stores in the L1. Thus, Racer naturally incorporates store-wait-free behavior (reordering of stores subject to synchronization constraints) with very low complexity.

Atomics. Atomic operations are the only operations that require coherence, i.e., guarantee the SWMR invariant. They cannot read an obsolete value, and therefore they should always bypass the L1 cache. At the same time, atomics need to guarantee atomicity in their load and store operations. For this, the LLC block that is read by an atomic operation must remain locked until the write is performed. Atomics perform a write-through to the LLC, once they modify the data. The write-through unlocks the block in the LLC.

Regarding reordering, RMW atomics that are composed by reads and writes, must also ensure load→load and store→store. Therefore, if the load conflicts, a self-invalidation operation must be performed in the cache of the requesting core. Similarly, before the atomic write-through is performed, all previous writes by that core should have been performed (or perform in Collapsed Order with the atomic write). The write-through of the atomics also inserts its address in the RAWR.

Thread migration. Racer is impervious to thread migration as the RAWR tracks writes not seen by other cores independent of which thread runs on them. If a thread migrates to another core (after draining its CSB by the OS), the new core may not "have seen" all the thread's writes. The resulting races with the thread's own writes cause self-invalidation and refetch of the thread's data in the new cache. On the other hand, the migrating thread may find the previous thread's conflicting write in its new cache without going through the RAWR. That is correct as the previous thread also drained its CSB before departing the core and all its writes (prior to and including the conflict) are visible to the migrating thread.

TABLE II
SYSTEM CONFIGURATION.

Parameter	Value
Processor frequency	3.0GHz
Block size / page size	64 bytes / 4KB
CSB (# of MSHRs per core)	64
Private L1 cache	32KB, 4-way
L1 cache access time	1 cycle
Shared L2 cache	512KB per bank, 16-way
L2 cache access time	Tag 6 cycles; tag+data 12 cycles
Memory access time	160 cycles
Network topology	2D mesh
Routing technique	Deterministic X-Y
Data / Control msg size	5 / 1 flits
Switch-to-switch time	6 cycles

VI. EVALUATION

A. Simulation environment.

Our evaluation is performed for a chip multiprocessor comprised of 64 in-order cores. The multiprocessor is modeled with the cycle-accurate GEMS [55] simulator, which offers a detailed timing model. GEMS is configured to use the GARNET [56] interconnection network implementation. The energy consumption values of the different structures of the multiprocessors have been obtained with the CACTI 6.5 tool [57], considering a 32nm process technology. Details of the simulated architecture are displayed in Table II.

In our evaluation, we employ a wide variety of applications from the Splash2 suite [58], with the standard inputs, and the PARSEC benchmark suite [59], with the *simmedium* input, except *Fluidanimate* and *Streamcluster*, which use the *simsmall* input due to simulation-time constraints. We simulate the entire application, but collect statistics only during its parallel phase.

B. Results

Synchronization detection accuracy. Data-race-free applications separate conflicting accesses by synchronization annotated in the code. Cache coherence protocols that implement an SC-for-DRF memory model must therefore self-invalidate the cache contents upon a synchronization operation with acquire semantics. However, in Racer, this synchronization is detected at runtime instead of statically annotated in the code. This can lead to less self-invalidation, for instance, when different cores execute critical sections protected by *different* locks.

Figure 8 shows the impact of the run-time race detection with *Racer* compared to *Static* annotation. The graph plots the number of self-invalidated blocks caused by synchronization, normalized to the *Static* approach (*Blackscholes* and *Swaptions* do not have synchronization operations). For Racer, we evaluate five different configurations. The first one, *Racer-Perfect-Word* employs a perfect Bloom filter that tracks written words. The second one, *Racer-Perfect-Line* employs a perfect Bloom filter that tracks written memory lines. The three remaining bars employ Bulk signatures [25] with 2048, 1024, and 512 bits per core. Since we model 64 cores, these sizes lead to RAWR structures of *16KB*, *8KB* and *4KB*, respectively, per LLC bank.

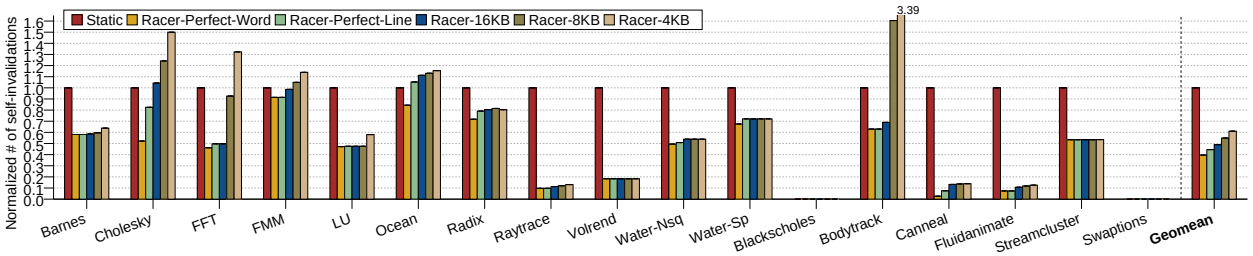

Fig. 8. Normalized number of self-invalidations

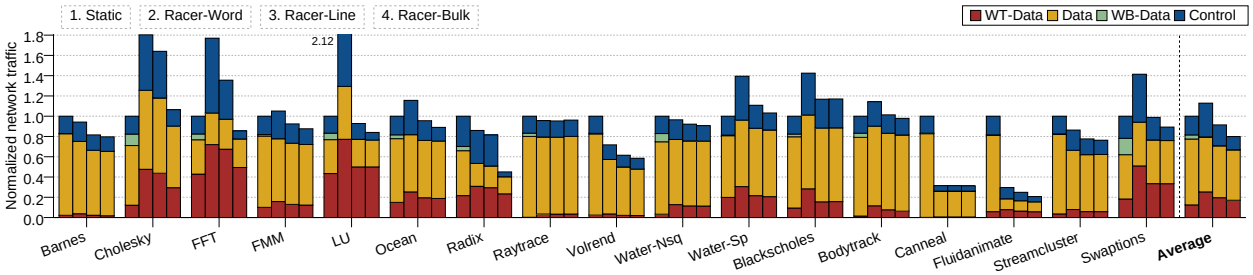

Fig. 9. Normalized network traffic

The first two bars give an idea of the potential of the runtime race detection performed by Racer. *Racer-Perfect-Word* invalidates the cache contents only when a real RAW race occurs during execution. Clearly, static self-invalidation leads to a large number of self-invalidated blocks. *Racer-Perfect-Line* includes the self-invalidated blocks due to *false-sharing* in the RAWR. Only a few applications, such as *Cholesky* and *Ocean*, are significantly affected by false-sharing, so we do not evaluate the fine granularity option further (RAWR and OSO work at cache-line granularity and false sharing causes superfluous self-invalidations).

The smaller the size of the signature is, the more blocks self-invalidate as a result of aliasing in the signature. We consider that a 16KB RAWR offers good trade-off between area and performance, reducing self-invalidations by 51.0% with respect to a static approach. Therefore, for the remaining of the evaluation we employ 16KB RAWRs.

Store-wait-free Effects. Stores in Racer write directly in L1 since they do not require write permission. The new value is however only visible to the local core until it is written-through to the LLC. During that time subsequent store operation to the same block can coalesce. In release consistency protocols such as VIPS-M [5] or QuickRelease [9] release operations are annotated in the code. Therefore, coalescing can be performed freely until a release operation is found. In Racer, each store can potentially be a release operation and coalescing in limited. In Section III-E, we propose several techniques to perform coalescing without violating store ordering as seen by other cores.

Figure 9 plots the traffic generated by the Racer's coalescing techniques normalized to the traffic generated by the *Static* case: a RC protocol with annotated acquire and release operations, inspired by VIPS-M. Traffic is split in four categories: i) all traffic generated by write-throughs of shared store operations; ii) data messages as a consequence of cache misses; iii) traffic generated by write-backs of private dirty data; iv) all control messages (aggregated). This section focus on the first category, where the statically annotated releases represent the ideal case.

Performing a write-through on every store operation (*Racer-Word*) can cause a large amount of traffic. Coalescing consecutive store operations to the same memory line (*Racer-Line*) reduces the traffic to acceptable levels for applications with high store spatial locality. However, some applications such as *Cholesky*, *FFT*, and *Radix* still require more sophisticated techniques. *Racer-Bulk* reduces the traffic due to write-throughs for these applications. By avoiding also unnecessary self-invalidations, the total traffic of *Racer-Bulk* is reduced by 20.1% compared to a static approach.

Write-through and Check-race Delay Effect. Figure 10 shows the effect in both execution time and network traffic of two design decisions in Racer: maximum write-through delay (left) and maximum check-race delay (right). We have tested several delay values from 50 cycles to 20000 cycles. The more frequent we perform the write-through or the check-race the more network traffic is generated by the coherence protocol. That is why we normalize network traffic to our more aggressive configuration (50 cycles). This extra traffic can increase network contention and therefore impact performance, as shown for low delay values. On the other hand, delaying the write-through or the check-race up to a large amount of cycles leads to slow propagation of new values for synchronization variables. This means, for instance, that a thread waiting to acquire a lock may wait for more time than necessary, e.g. when the lock is actually free.

Our analysis shows that delay values between 500 cycles and 2000 cycles achieve the best performance without generating a significant amount of traffic overhead with respect to the configuration with lowest traffic requirements but also with worst performance (20000 cycles). Based on this numbers, we

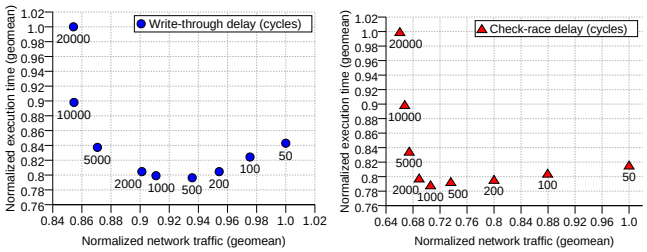

Fig. 10. Effect of write-through and check-race delays

choose to force a write-through when a block has been locally written in L1 for 1000 cycles and we issue a check-race when a block has been in the L1 for 1000 cycles.

Performance. Having analyzed the impact of the RAWR size, the store coalescing approaches, and the delay of write-through and check-race events, this section focuses on the performance advantages of Racer and its prediction policy. Figure 11 shows the execution time of the applications for five different protocols: a directory protocol with *MESI* states that guarantees SC; a directory protocol that guarantees TSO (*MESI-TSO*) as described by Culler et al. [60], i.e., that allows load operations to bypass store operations; a protocol that guarantees SC-for-DRF exposing synchronization to the hardware (*VIPS-M*); Racer with a 16KB RAWR and Bulk coalescing; and Racer enhanced with race-detection prediction. Results are normalized to *MESI*.

The relaxed store→load order in *MESI-TSO* enables performance improvements of 12.5%. Statically exposing synchronization to hardware (VIPS-M) can benefit some applications, but some others like *Fluidanimate* or *Canneal* experience dramatic slow-downs compared to write-invalidation in *MESI*. Run-time race detection can considerably reduce execution time in the applications where the overhead of static self-invalidation is significant. For example, *Fluidanimate* employs a large amount of locks which require self-invalidation in *VIPS-M* on each acquire, but in *Racer* they do not self-invalidate if they are to different lock variables. A similar case happens in *Canneal* with atomic operations. In other applications such as *Barnes*, *FMM*, and *Streamcluster*, *Racer* performs worse than *VIPS-M*. In these applications a low value (e.g., 50 cycles) for the delay write-through in *Racer* performs better than the selected 1000 cycles threshold. This applications will benefit from the prediction mechanism. Overall, *Racer* helped with prediction (*Racer-Hint*) is able to achieve similar performance as a non-scalable *MESI-TSO* protocol and better performance than *VIPS-M*, while also providing stronger consistency.

Energy consumption. We measure the energy consumption of the evaluated protocols (Figure 12). We normalize again to *MESI*, and we account for the consumption of the TLBs, L1 caches, network, LLC, and RAWR+OSO. First, the use of virtual caches in request-response protocols based on self-invalidation can mitigate the energy consumption of the TLB, since it is only accessed on L1 misses [6]. Second, the network traffic and LLC accesses are reduced compared to protocols that expose synchronization to hardware, since the number of invalidated blocks, and consequently, L1 misses is reduced. All in all, *Racer* improves energy consumption by 21.5% compared to *MESI* and by 3.5% compared to *VIPS-M*. *Racer-Hint* allows the use of larger leases for blocks cached in L1 and stores performed in the L1, without sacrificing performance, and therefore, traffic and LLC accesses are further reduced, thus reducing the overall energy consumption (26.4% compared to *MESI* and 9.5% compared to *VIPS-M*).

C. Area considerations and scalability.

Racer adds two main structures to guarantee store order (OSO) and to detect races (RAWR). OSO employs a counter and a small bloom filter per core. The counter needs to account for the number of acks received, whose maximum and minimum is bounded by the maximum number of write-through blocks in a Bulk operation (number of MSHRs). Therefore we require $\log_2(64 \times 2) = 7bits$. The size of the bloom filters in OSO is 128 bits. This gives an area of approximately 1KB. On the other hand the (evaluated) size of the RAWR is 16KB. The prediction structures and the self-invalidation counters in Racer are very small compared to these structures acounting for less than 1KB, so the area required for a 64-core system is less than 18KB per node. Considering $1\times$ coverage for L1 caches, as assumed in this evaluation, the size of a directory cache (64 bits per directory entry) is 6.1KB per node (12.2KB for $2\times$ coverage).

While, in this specific case, Racer's area is larger (but in the same vicinity) than a typical directory cache, Racer's advantage lies in *scalability*. The argument is directly analogous to the argument made for TagLess Directories [28] that use a similar signature-based organization but replicate the functionality of a directory (explicit invalidation and indirection). In short, as in [28], the *total* size of the RAWR (aggregate capacity of all banks) increases linearly with the number of cores, keeping a constant *expected number of false-positive bits* (in contrast to a typical *full-map* directory that increases quadratically). This means that in a distributed RAWR (one bank/core), the size of RAWR bank remains *constant*. As the number of cores doubles, the width of the RAWR bank is *halved*, keeping the total signature (across all banks) constant and therefore, maintaining its accuracy. Only when the width of a RAWR bank becomes very small its hashing efficiency degrades at which point the total signature must be increased. See [28] for a model and an extensive analysis of the scaling of signatures.

VII. Conclusions

In this paper we propose an alternative approach to TSO relying on *simple* coherence mechanisms found in SC-for-DRF coherence protocols: self-invalidation and buffered (coalesced) write-through. An SC-for-DRF protocol using these mechanisms requires software cooperation, and in particular, exposing program synchronization to the hardware. Unfortunately, this is not always possible, or easy, or complete. Our goal in

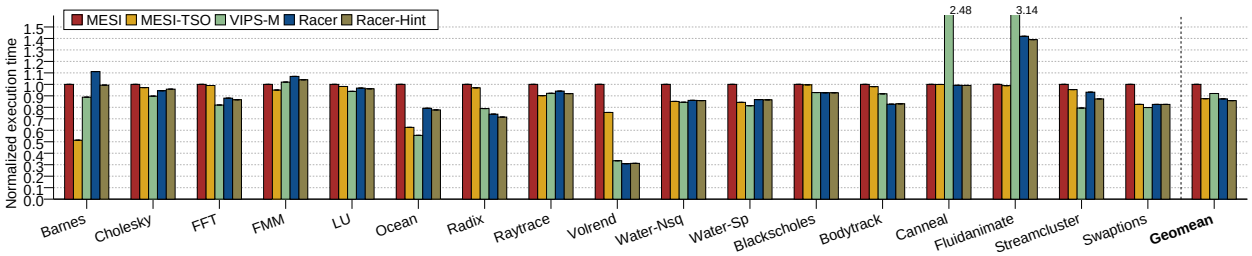

Fig. 11. Normalized execution time

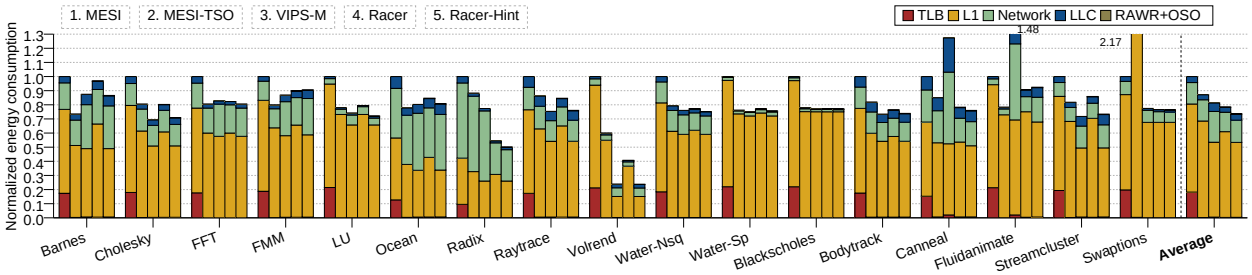

Fig. 12. Normalized energy consumption

this work is to apply these mechanisms without any software involvement.

We achieve this by dynamically detecting read-after-write races in a program. The underlying concept is to equate detected races to release-acquire synchronization of the program. Thus, a detected acquire causes self-invalidation and a detected release must exhibit a valid TSO behavior.

Our approach melds the simplicity of the SC-for-DRF coherence mechanisms with a strong consistency model (TSO), but at the same time relaxes load→load and store→store orderings when these cannot be detected. It exceeds the performance of an SC-for-DRF coherence protocol driven by acquire-release annotations in the program, while at the same time lowers energy consumption in the cache hierarchy and network even further.

Acknowledgment

This work has been jointly supported by the "Fundación Séneca – Agencia de Ciencia y Tecnología de la Región de Murcia" under the project "Jóvenes Líderes en Investigación" 18956/JLI/13, the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2015-66972-C5-3-R, and the Swedish VR (grant no. 621-2012-5332).

References

[1] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture, M. D. Hill, Ed. Morgan & Claypool Publishers, 2011.

[2] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers (TC)*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[3] 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 *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 155–166.

[4] T. J. Ashby, P. Díaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Transactions on Computers (TC)*, vol. 60, no. 4, pp. 472–483, Apr. 2011.

[5] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[6] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *40th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2013, pp. 535–547.

[7] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient hardware support for disciplined non-determinism," in *18th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2013, pp. 13–26.

[8] I. Singh, A. Shriraman, and W. W. L. Fung, "Cache coherence for gpu architectures," in *19th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2013, pp. 578–590.

[9] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 189–200.

[10] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *14th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2014, pp. 427–440.

[11] A. Ros, M. Davari, and S. Kaxiras, "Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies," in *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 186–197.

[12] A. Ros and S. Kaxiras, "Callback: Efficient synchronization without invalidation with a directory just for spin-waiting," in *42nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2015, pp. 427–438.

[13] H. Sung and S. V. Adve, "DeNovoSync: Efficient support for arbitrary synchronization without writer-initiated invalidations," in *15th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2015, pp. 545–559.

[14] M. Elver and V. Nagarajan, "RC3: Consistency directed cache coherence for x86-64 with RC extensions," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 292–304.

[15] S. V. Adve and M. D. Hill, "Weak ordering – a new definition," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.

[16] ISO, *ISO/IEC 14882:2015 Information technology — Programming languages — C++*. International Organization for Standardization, 2015.

[17] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 2014.

[18] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, 2011.

[19] A. Ros and S. Kaxiras, "Fast&furious: A tool for detecting covert racing," in *6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA) and 4th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (DITAM)*, Jan. 2015, pp. 1–6.

[20] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016.

[21] 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," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[22] C. von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu, "Conditional memory ordering," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 41–50.

[23] K. Gniady, B. Falsafi, and T. Vijaykumar, "Is SC + ILP = RC?" in *26th Int'l Symp. on Computer Architecture (ISCA)*, May 1999, pp. 162–171.

[24] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *34th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 266–277.

[25] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk disambiguation of speculative threads in multiprocessors," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 227–238.

[26] P. E. McKenney, "Memory ordering in modern microprocessors, part i & part ii," *Linux Journal*, vol. 30, no. 137, pp. 52–57, Sep. 2005.

[27] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "An efficient, self-contained, on-chip, directory: DIR_1-SISD," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 317–330.

[28] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 423–434.

[29] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, "SPATL: Honey, i shrunk the coherence directory," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2011, pp. 148–157.

[30] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011, pp. 93–103.

[31] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *18th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2012, pp. 129–140.

[32] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013, pp. 359–370.

[33] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for tso," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 165–176.

[34] X. Yu and S. Devadas, "Tardis: Time traveling coherence algorithm for distributed shared memory," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 227–240.

[35] G. Kurian, Q. Shi, S. Devadas, and O. Khan, "Osprey: Implementation of memory consistency models for cache coherence protocols involving invalidation-free data access," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 392–405.

[36] L. Lamport, "Times, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[37] A. Ros and A. Jimborean, "A dual-consistency cache coherence protocol," in *29th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2015, pp. 1119–1128.

[38] ——, "A hybrid static-dynamic classification for dual-consistency cache coherence," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. PP, no. 99, Feb. 2016.

[39] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 184–195.

[40] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 501–512.

[41] Y. Li, R. G. Melhem, and A. K. Jones, "Practically private: Enabling high performance cmps through compiler-assisted data classification," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 231–240.

[42] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *39th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2012, pp. 524–535.

[43] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *42nd Int'l Conf. on Parallel Processing (ICPP)*, Oct. 2013, pp. 562–571.

[44] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 111–122.

[45] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "Tokentlb: A token-based page classification approach," in *Int'l Conf. on Supercomputing (ICS)*, Jun. 2016, pp. 26:1–26:13.

[46] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[47] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[48] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based transactional memory," in *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2006, pp. 254–265.

[49] G. K. Konstadinidis, H. P. Li, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. D. Lin, P. Loewenstein, H. Park, V. Srinivasan, D. Huang, C. Hwang, W. Hsu, C. McAllister, J. Brooks, H. Pham, S. Turullols, Y. Yanggong, R. Golla, A. P. Smith, and A. Vahidsafa, "SPARC M7: A 20 nm 32-core 64 MB L3 cache processor," *IEEE Journal of Solid-State Circuits*, Jan. 2015.

[50] J. Oh, M. Prvulovic, and A. Zajic, "TLSync: Support for multiple fast barriers using on-chip transmission lines," in *38th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011, pp. 105–116.

[51] B. M. Beckmann and D. A. Wood, "TLC: Transmission line caches," in *36th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2003, pp. 43–54.

[52] M.-C. F. Chang, J. Cong, A. Kaplan, C. Liu, M. Naik, J. Premkumar, G. Reinman, E. Socher, and S.-W. Tam, "Power reduction of CMP communication networks via RF-interconnects," in *41th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Nov. 2008, pp. 376–387.

[53] T. Krishna, A. Kumar, P. Chiang, M. Erez, and L. shiuan Peh, "NoC with near-ideal express virtual channels using global-line communication," in *16th HotInterconnects Symp.*, Aug. 2008, pp. 11–20.

[54] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, Apr. 1996.

[55] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[56] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[57] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0," HP Labs, Tech. Rep. HPL-2009-85, Apr. 2009.

[58] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[59] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[60] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.