Peacenik: Architecture Support for Not Failing under Fail-Stop Memory Consistency

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

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

Vignesh Balaji
Carnegie Mellon University, USA
vigneshb@andrew.cmu.edu

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

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

Abstract

Modern shared-memory systems have erroneous, undefined behavior for programs that are not well synchronized. A promising solution is to provide *fail-stop memory consistency*, which ensures well-defined behavior for *all* programs. While fail-stop consistency avoids undefined behavior, it can lead to unexpected failures, imperiling performance or progress.

This paper presents architecture support called *Peacenik* that avoids failures in the context of fail-stop memory consistency. We demonstrate Peacenik by applying Peacenik's general mechanisms to two existing architectures that provide fail-stop consistency. A simulation-based evaluation shows that Peacenik eliminates nearly all of the high costs of fail-stop behavior incurred by the baseline architectures, demonstrating how to get the benefits of fail-stop consistency without incurring most or all of its costs.

CCS Concepts. • **Computer systems organization → Reliability**; **Availability**; **Processors and memory architectures**; **Multicore architectures**; • **Software and its engineering** → *Consistency*.

Keywords. Data races, fail-stop memory consistency, conflict exceptions, failure avoidance

ACM Reference Format:
Rui Zhang, Swarnendu Biswas, Vignesh Balaji, Michael D. Bond, and Brandon Lucia. 2020. Peacenik: Architecture Support for Not Failing under Fail-Stop Memory Consistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3373376.3378485

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378485

1 Introduction

To achieve high performance, today's shared-memory systems use compiler and hardware optimizations that assume *data-race-free* (DRF) programs. The DRF assumption is codified in memory models including the C/C++ and Java memory models, which ascribe undefined or ill-defined semantics to programs with data races [3, 4, 13–15, 43]. Data races are not just erroneous in theory, but can cause serious problems in practice [36, 37, 48, 65]. Mature, widely used applications have data races, which are not easy to detect or eliminate fully [23, 27, 34–37, 40, 48, 65]. Production systems typically have different run-time characteristics than testing environments, making it difficult to catch all production-time data races during in-house testing.

Researchers have sought to provide stronger *end-to-end* (i.e., with respect to the source program) memory consistency by restricting compiler and hardware optimizations—notably by providing end-to-end SC [5, 58, 61, 64]. (Restricting only hardware [16, 20, 26, 38, 39, 52] or the compiler [45] fails to provide *end-to-end* SC.)

Alternatively, instead of restricting optimizations, a system can detect the *conditions* for weak or undefined semantics and treat them as *fail-stop errors* [9, 10, 21, 25, 41, 44, 53, 60, 67]. Under *fail-stop memory consistency*, all programs have well-defined behavior, and an execution stops if it might have undefined behavior due to a data race.

Fail-stop consistency solves one problem (undefined, erroneous behavior) but introduces another: data races can lead to unexpected production-time failures [23, 27, 35, 40]. If a system handles such failures by rebooting (i.e., terminating and restarting) a program or a component, fail-stop errors translate into more execution time.

This paper's goal is to avoid failures due to fail-stop consistency as much as possible, with minimal cost and complexity while retaining strong consistency guarantees. To that end, we introduce *Peacenik*, a set of architecture mechanisms

that extend a baseline fail-stop consistency design to provide comprehensive, best-effort support for avoiding failures. Peacenik's main components are support for (1) *pausing* a core's execution to avoid conflicts that lead to failures, (2) *restarting* regions of code that would otherwise fail, and (3) *cache replacement policy modifications* that increase the chances of a region being eligible for restart. Peacenik cannot avoid all failures because its mechanisms are best-effort—in particular, a core cannot restart a region if the dirty data written by the region escapes from its private caches due to eviction or a remote coherence request—but Peacenik's mechanisms are effective in practice. While Peacenik employs mechanisms borrowed from hardware transactional memory (HTM), its design differs from both best-effort and unbounded HTM (Section 2).

We show the generality and effectiveness of Peacenik by developing prototype implementations that extend *ARC* and *Conflict Exceptions* (CE), two architectures from prior work that provide fail-stop consistency [10, 41]. While ARC and CE—like any system that supports precise, unbounded region conflict detection—incur significant complexity on their own, Peacenik's architecture support for avoiding failures is relatively simple and incurs low costs and complexity on top of any underlying system.

A simulation-based evaluation shows that, under fail-stop consistency semantics *without Peacenik*, several benchmarks and real-world programs suffer from fail-stop errors due to data races. Our experiments model these errors by rebooting the program, or by repeating a request in the case of server programs, leading to high performance costs from fail-stop errors. We also find that fail-stop errors incur significant costs even if the experiments model idealized process checkpointing. In contrast, Peacenik avoids nearly all fail-stop errors in our experiments, achieving essentially the same performance as data-race-free programs would achieve, while also providing strong behavior guarantees. These results show that simple, effective failure-avoidance mechanisms can overcome the primary obstacle to practical fail-stop memory consistency.

2 Background, Problem, and Related Work

This section overviews fail-stop consistency and its costs, and motivates the importance and challenge of avoiding failures.

2.1 Fail-Stop Memory Consistency

As Section 1 explained, data races are erroneous and lead to serious failures [3, 4, 13–15, 36, 37, 43, 48, 65]. Researchers have proposed systems that provide *fail-stop semantics* by generating a *consistency exception* upon a data race [19, 21, 24, 49, 53, 67]. Since sound and precise dynamic race detection is expensive [19, 24], efficient systems provide fail-stop consistency by detecting *conflicts between concurrently executing synchronization-free regions (SFRs)* [9, 10, 25, 41, 44, 60]. (An SFR is a sequence of per-thread dynamic instructions demarcated by synchronization operations.) In such systems—this paper's focus—a conflict indicates a data race, while a conflict-free execution is equivalent to a serialization of executing SFRs. The rest of this paper uses the terms "SFR" and "region" interchangeably.

Lazy and eager conflicts. In this paper, an important distinction exists between conflicts that are detected *lazily* versus *eagerly*. We define a *lazily detected conflict*, or a *lazy conflict*, as a conflict detected when it is "too late" to ensure serializability of regions, i.e., if the continued execution cannot guarantee region serializability. In contrast, an *eagerly detected conflict*, or an *eager conflict*, is detected when it *may still be possible* for the execution to achieve region serializability. Note that, in contrast with typical definitions of eager conflict detection [29], our definition of an eagerly detected conflict permits both accesses to have already executed, so long as serializing the conflicting regions is still feasible. Our Peacenik design handles both lazy and eager conflicts.

Existing fail-stop consistency systems. *DRFx* is a system that provides fail-stop consistency by checking for conflicts between regions that are *bounded* (statically limited to a fixed number of memory accesses) [44, 60]. Bounded regions reduce DRFx's hardware requirements but require compiler modifications that limit cross-region optimization. DRFx detects conflicts lazily by broadcasting an ending region's read and write sets to other cores, which check for conflicts.

Valor, *Conflict Exceptions* (CE), and *ARC* detect conflicts between (unbounded) SFRs, i.e., regions bounded only at synchronization operations [9, 10, 41]. The resulting memory consistency model, called *SFRSx*, ensures that every execution is SFR serializable or generates a consistency exception due to a data race. *Software or architecture support alone* is sufficient to provide end-to-end SFRSx: hardware and compilers already restrict optimizations across SFR boundaries and thus require no complementary modifications.

Valor provides SFRSx in software only, slowing programs by almost 2× on average [9]. CE and ARC are architectures that provide SFRSx [10, 41]. CE detects all conflicts eagerly, while ARC detects some conflicts eagerly and others lazily.

By adding simple hardware components, Peacenik's mechanisms (Section 3) and architecture design (Section 4) can extend any system that provides fail-stop behavior for conflicting regions, including DRFx, Valor, CE, and ARC. Our prototype implementations of Peacenik extend CE and ARC (Section 4).

2.2 Failures Impact Performance

Fail-stop consistency semantics provide a solid theoretical foundation for memory consistency [3]. But what are the practical implications of fail-stop consistency? What happens when a program encounters a consistency exception?

	Best-effort HTM	Unbounded HTM	Failure avoidance-mechanisms under fail-stop consistency
A conflict is a(n) …	Misspeculation	Misspeculation	Error
Conflict detection …	May be imprecise	May be imprecise	Must be precise
Pause execution at conflicts?	No, typically	Yes, for bounded time	Yes, indefinitely
Regions may become ineligible for restart?	Yes	No	Yes
Action when regions become ineligible for restart	Abort	N/A	Continue execution

Table 1. Comparison between the requirements of HTM and failure-avoidance mechanisms.

Although a program could potentially *catch and handle* a consistency exception using language or library support, we envision two major problems with that approach:

- Like memory errors and null pointer exceptions, consistency exceptions are unexpected. Writing handlers for every potentially shared access would be a huge programming burden.
- Existing systems that provide fail-stop semantics attain efficiency by detecting some or all conflicts *after* both conflicting accesses have already executed [9, 10, 44, 60], making *precise exceptions* infeasible.[1]

Instead, we envision systems handling a consistency exception as a termination signal, by *rebooting* (i.e., terminating and restarting) the program.

Fail-stop behavior essentially trades one problem for another, avoiding the subtle failure modes of data races, but causing systems to stop executing and repeat some or all of their work upon encountering a data race. Consequently, consistency exceptions represent a threat to performance: a program that generates consistency exceptions will take longer to produce results or handle requests, wasting time and other resources. In this work, we seek to limit such costs, while still providing well-defined behavior for all program executions.

2.3 Avoiding Failure under Fail-Stop Semantics

Failure-avoidance mechanisms arguably should be implemented in *hardware*. Supporting failure avoidance architecturally permits the failure avoidance mechanism access to microarchitectural state such as the access metadata associated with private and shared cache lines used by existing fail-stop systems [10, 41, 44, 60].

Some prior work avoids failures with software support. *Avalon* extends software-based SFRSx support to avoid consistency exceptions by pausing threads that detect conflicts [71]. *SOFRITAS* enforces conflict serializability using two-phase locking, effectively pausing conflicting threads [18]. Both approaches rely on software *only* and do not introduce architecture support for pausing or resuming cores, nor can they support restarting regions.

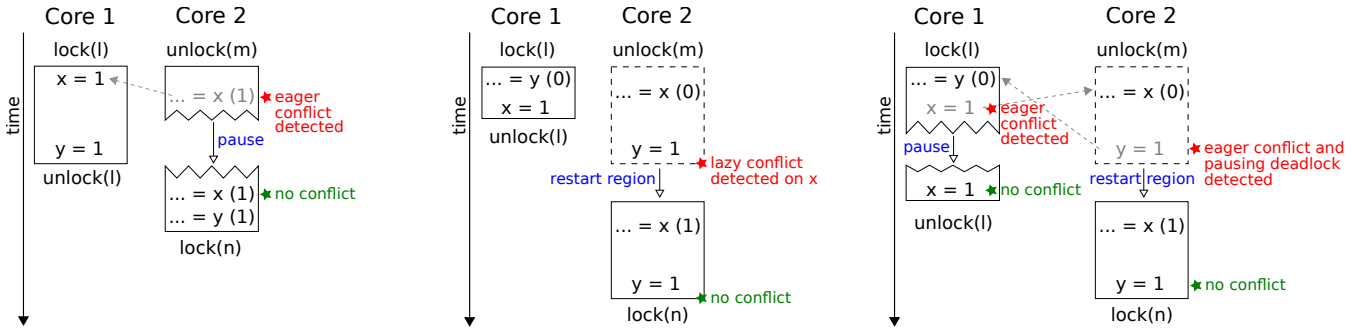
Comparison with HTM. Architecture support for failure avoidance under fail-stop consistency is related to hardware transactional memory (HTM) [6, 11, 12, 28, 29, 31, 66, 69]. While our Peacenik design borrows and adapts some mechanisms from HTM, Peacenik differs from both best-effort and unbounded HTM as a consequence of targeting the fail-stop consistency context. Table 1 compares Peacenik's requirements with those of best-effort and unbounded HTM. HTM conflict detection is generally imprecise because a conflict is only a misspeculation, while failure avoidance must be precise since conflicts are errors. Some HTMs support pausing and detecting pausing cycles (e.g., DATM [51] and Wait-n-GoTM [32]); HTM must not impede progress due to conflicts, while Peacenik can pause execution at a conflict indefinitely since conflicts are errors. Likewise, HTM must support restarting a region on a conflict—best-effort HTM aborts a region that becomes ineligible for restart due to escaped dirty values written by the region, while unbounded HTM ensures regions are restartable—but Peacenik allows regions to continue execution even if they become ineligible for restart.

Both best-effort and unbounded HTM have limitations as potential failure-avoidance mechanisms. Best-effort HTM provides no progress guarantee and in practice cannot complete long transactions or transactions with irrevocable operations. Unbounded HTM adds significant complexity and cost compared with this paper's Peacenik mechanisms.

Other related work. Some prior work applies HTM or HTM-like mechanisms to *enforce* serializability of SFRs, incurring the complexity of unbounded HTM [5, 28, 47, 55, 56].

Wait-n-GoTM and *BulkSMT* use pausing to avoid conflicts [32, 50]. Like Peacenik, *BulkSMT* supports pausing in a non-speculative setting, but provides a consistency guarantee only with respect to the compiled program (serializability of bounded chunks), not an end-to-end guarantee [50]. *SC-safe* detects dependency cycles due to SC violations or false sharing, in order to enforce SC and log violations of SC [20]. In contrast, Peacenik detects deadlocks that it has introduced by pausing at detected region conflicts. *ConAir* and *BugTM*

[1] A precise exception is generated immediately before a conflicting access executes. Note that the conditions for a precise exception (detecting a conflict before one of the conflicting accesses has executed) are stricter than the conditions for an eagerly detected conflict (as defined in Section 2.1).



(a) Peacenik avoids a consistency exception at an eagerly detected conflict by pausing Core 2.

(b) Peacenik avoids a consistency exception at a lazily detected conflict by restarting Core 2's region.

(c) Peacenik detects a pausing deadlock caused by two eagerly detected conflicts, and avoids a consistency exception by restarting Core 2's region.

Figure 1. Three examples showing how Peacenik's mechanisms avoid consistency exceptions. For (a) and (c), we assume the conflicts are detected *eagerly*, while (b) assumes the conflicts are detected *lazily*. In all examples, shared variables x and y are initially 0. Boxes demarcate synchronization-free regions (SFRs); dashed boxes indicate restarted regions. Gray text indicates instructions delayed due to pausing. Gray dashed arrows indicate eager conflicts.

re-execute code regions to recover from concurrency bug failures such as atomicity violations [17, 72].

How should failure-avoidance mechanisms be used? In *testing*, it is counterproductive to hide errors, so developers should *disable* failure-avoidance mechanisms. However, complete testing is infeasible. Even after fixing known data races (e.g., races detected during testing), large concurrent programs are likely to have unknown data races that manifest rarely and unexpectedly on *production* systems. Production systems should *enable* failure-avoidance mechanisms to avoid or reduce real costs from fail-stop errors. To aid debugging, a failure-avoidance mechanism such as Peacenik can log conflicts avoided in production and report them to developers.

3 Peacenik Design Overview

This section overviews the design of Peacenik, a set of architecture mechanisms for avoiding consistency exceptions. Section 4 describes Peacenik at an architectural level and studies two prototype implementations of Peacenik by extending existing fail-stop systems.

Pausing at region conflicts. Existing fail-stop systems generate a consistency exception when they detect or infer a region conflict between executing cores,[2] which occurs when one core's memory access conflicts with a memory access made in another core's concurrent region [9, 10, 41, 44, 60]. Figure 1(a) shows an example of a region conflict. In the example, the system eagerly detects a write–read conflict at Core 2's read of x with Core 1's write to x. A system that ensures region serializability must not allow Core 2 to continue execution, at least until Core 1's region completes; if Core 2 were to read y immediately after reading x (and see 0), the execution would become unserializable.

Peacenik avoids the region conflict by *pausing* Core 2's execution until Core 1's ongoing region completes, when Core 2's access can then proceed without any conflict. Peacenik pauses Core 2 at the conflicting access until Core 1 finishes its ongoing region, serializing Core 2's region after Core 1's.

Pausing is not a panacea for avoiding consistency exceptions, for two reasons. First, Peacenik cannot usefully pause a core at a conflict detected *lazily* because it is too late for pausing to ensure region serializability (Section 2.1). In Figure 1(b), the system detects a conflict on x lazily at the end of Core 2's region: the core has read an out-of-date value for x, and it is too late to pause to avoid violating serializability.

Second, pausing can lead to a *pausing deadlock* when multiple cores are involved in a cycle of wait-for dependencies. Figure 1(c) shows that two eagerly detected read–write conflicts lead to a pausing deadlock when both cores try to avoid the conflicts by pausing. Peacenik detects a pausing deadlock by detecting a cycle of wait-for dependencies, and selects one core involved in the deadlock to generate a consistency exception (or to restart its region if eligible, as discussed next).

Restarting regions. Peacenik further avoids consistency exceptions by *restarting eligible regions* that would otherwise generate a consistency exception (due to a lazily detected conflict or a pausing deadlock). By restarting a region *R*, Peacenik tries to serialize *R* after any other core's region with which *R* conflicted. Region restart is best effort; a region is eligible to restart if no data written by the region has become

[2] For now we assume each core executes a single pinned thread. We relax this limitation and discuss Peacenik's corresponding handling in Section 4.6.

visible to reads by other cores (e.g., through a cache eviction or remote coherence request).

Figures 1(b) and 1(c) show how restarting a region avoids a consistency exception due to a lazily detected conflict and a pausing deadlock, respectively. In both examples, Peacenik avoids the consistency exception by restarting Core 2's ongoing region (the dashed box) and invalidating privately cached data, serializing it after Core 1's completed region. Both examples assume that the regions are eligible for restart.

Regions can repeatedly restart (e.g., by repeatedly conflicting with each other), leading to livelock. As a simple extension to the above design, Peacenik could avoid livelock by using exponential backoff and by restarting only the youngest eligible region.

Enhancing best-effort region restart. To improve the odds that a region will be eligible to restart, Peacenik modifies the replacement policy of cores' private caches. The modified policy avoids evicting lines written by the current region and instead favors evicting other lines in the same associativity set.

Fallback. In the case of a pausing deadlock or lazy conflict in which no region is eligible to restart, Peacenik falls back to a consistency exception.

4 Peacenik Architecture Design

This section introduces a Peacenik architecture design that generally applies to any baseline architecture that provides fail-stop consistency. We describe how to apply the Peacenik architecture design to two existing fail-stop architectures, ARC and CE [10, 41]. The resulting designs are called *ARC+Peacenik* and *CE+Peacenik*. This section assumes cores with two-level private caches, i.e., L1 and L2. We describe quantitatively how Peacenik adds minimal complexity on top of an underlying system that provides fail-stop consistency.

4.1 Background: ARC and CE Details

Both ARC and CE are architecture designs that provide SFRSx (Section 2.1) by detecting conflicts between concurrently executing regions [10, 41]. ARC detects conflicts both eagerly and lazily, while CE detects all conflicts eagerly.

ARC. In ARC, cores execute synchronization-free regions (SFRs) mostly independently. At a high level, when a region completes, ARC uses a *region commit protocol* to ensure that the region's memory accesses did not conflict with accesses in a region executed by another concurrently executing core, and that the region read consistent memory state. The protocol consists of three steps: (1) *pre-commit*, which writes back access metadata about privately cached dirty lines to the LLC, detecting eager conflicts in the process; (2) *read validation*, which compares privately cached read-from lines with the LLC's versions, detecting lazy conflicts for inconsistent lines and eager write–read conflicts; and (3) *post-commit*, which writes back privately cached dirty lines to the LLC and clears access metadata for the core. When a core evicts a private cache line read and/or written by its ongoing region, ARC immediately performs read validation and/or pre-commit for the evicted line only.

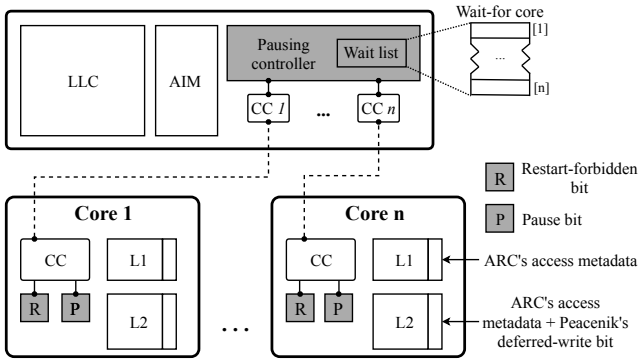


Figure 2. ARC+Peacenik architecture components. Additions to ARC are shaded; Peacenik also adds a per-line *deferred-write bit* to the L2 access metadata. Components are not shown to scale, e.g., the pausing controller's actual area requirements are negligible: logic for maintaining the wait list, which consists of n wait-for core IDs.

For all detected conflicts, ARC delivers a consistency exception to the involved core, which handles the exception as a termination signal. Before performing an externally visible operation such as a system call, ARC (and other fail-stop systems that detect conflicts lazily) must provide timely exceptions. In particular, ARC preemptively performs read validation to check for conflicts before an externally visible operation.

Architecturally, ARC has three main components, illustrated by Figure 2 (excluding the shaded components, which Peacenik adds): (1) per-line metadata in each core's private L1 and L2 caches, (2) an *access information memory* (AIM), and (3) distributed *consistency controllers* (CCs). The metadata added to private cache lines consists of two bits per byte that record whether the core has read and written each byte in the line. The AIM is a banked cache (banking not shown in Figure 2) of each core's access metadata for each line resident in the LLC. A core's *core-side* and *AIM-side* CCs manage the movement of data and metadata between the core, LLC, and AIM.

CE. CE introduces per-byte local and remote access bits to each privately cached line to keep track of bytes accessed by the ongoing region of the local core and remote cores, respectively. It piggybacks on MESI coherence messages to communicate access information indicated by the access bits between cores, and checks for region conflicts before performing any access by comparing the local access 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 message to other cores at each region boundary, and a core receiving an end-of-region message clears corresponding remote bits in its own private caches and sends back an acknowledgment message. 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*.

A CE architecture diagram (not shown) would differ from Figure 2 by removing the AIM and CCs, adding a directory and core-to-core communication, and using different access metadata than ARC.

Peacenik's architectural extensions can be added to a variety of architectures, including multi-socket NUMA systems, and various cache coherence protocols. The Peacenik prototype implementations that extend ARC and CE are a single-socket design with a single shared cache (LLC).

4.2 Architecture Support for Pausing

To supporting pausing, Peacenik introduces (1) a *pausing controller* that maintains logic to determine when a core should pause and (2) a per-core *pause bit* that indicates that a core is paused. When the baseline fail-stop architecture (e.g., ARC or CE) detects a conflict eagerly, instead of triggering a consistency exception, it signals the pausing controller. The baseline architecture must identify two cores to the pausing controller: a *paused core* that is about to pause and a *wait-for core* that the paused core is waiting on. The pausing controller records the wait-for relationship between the two cores in a structure called the *wait list*. The wait list can be implemented as a simple lookup table with one entry per core, because each paused core waits directly on at most one other core at a time.

The pausing controller also sets the paused core's pause bit, by activating a dedicated control signal to the core. When a core's pause bit is set, the core stops executing instructions. A pipelined core stalls its pipeline, and flushes its speculatively executed instructions from the pipeline. A paused core's cache controller continues responding to cache line requests from other cores and the LLC.

When a wait-for core completes its ongoing region, it signals the pausing controller to clear the paused core's pause bit and entry from the wait list. Once the pausing controller resets a core's pause bit, the core resumes execution.[3]

The pausing controller detects a *pausing deadlock*, which is a dependency cycle between a set of paused cores, by performing deadlock detection [46].

[3]Each core could have a bit indicating whether it is a wait-for core. To avoid a race between setting the wait-for bit and the relevant region ending, the design could identify regions uniquely (e.g., with per-core counters). Alternatively, instead of using per-core wait-for bits, the pausing controller could periodically wake up each paused core, which would recheck the conflict and the wait-for relationship.

To implement support for pausing in ARC and CE, it is straightforward to add a centralized pausing controller to them, as shown in Figure 2 for ARC. The pausing controller could instead be distributed; however, it is an unlikely bottleneck since it is signaled rarely: when a conflict is detected or a wait-for core ends a region. Arbitration at the pausing controller is simple because of the rarity of these events. At small core counts, a bus connecting all cores to the pausing controller may be sufficient. At larger core counts, an ordered buffer at the pausing controller that NACKs signals on overflow is sufficient. After such a NACK, a core could try again, or eventually give up on pausing and terminate with an exception.

Pausing in ARC+Peacenik. When a core's AIM-side CC detects a conflict eagerly, it signals the pausing controller to set the paused core's pause bit. When the wait-for core completes post-commit for its ongoing region, it signals the pausing controller, which clears the paused core's pause bit, allowing the core to resume.

In ARC, there are two situations in which a core may detect a conflict eagerly and consequently pause: during region commit and on an eviction from a core's L2 cache. During region commit, a core's CC may detect an eager conflict when performing pre-commit or read validation, in which case the core's CC pauses the core; it restarts the commit protocol when the core resumes. Read validation may detect a lazy conflict, which is unavoidable by pausing, triggering a region restart if eligible. Likewise, a core's CC may detect a conflict when performing pre-commit or read validation on a single line evicted from the private L2; it handles the conflict similarly to the commit protocol's handling, except there is no need to restart the protocol.

Pausing in CE+Peacenik. When a core's private cache detects a conflict before accessing a cache line, it signals the pausing controller to pause the core. The pausing controller then checks the line's local bits in each of the other cores' private caches (or the global table if the line was evicted by a core) and identifies a core as the *wait-for core* if the core has conflicting local bits set with the pause core. When the wait-for core ends its ongoing region, clearing its local access bits and receiving all acknowledgments from other cores (a paused core still acknowledges another core's end-of-region message), the wait-for core signals the pausing controller to clear the paused core's pause bit, allowing the core to resume.

4.3 Architecture Support for Region Restart

When the baseline fail-stop architecture detects a conflict lazily or when two or more cores enter into a pausing deadlock, Peacenik tries to avoid a consistency exception by restarting the execution of a relevant core's ongoing region that is *eligible* to restart. A core's region is eligible to restart if, during the region's execution, the core has not written

back any line updated by the region from its private cache(s) to the LLC or any private cache of another core. A core becomes ineligible to restart when it writes back such a line, making it visible to other cores in the system.

Peacenik introduces a per-core *restart-forbidden bit* that indicates whether a core's region is eligible to restart (Figure 2). Whenever a core writes back a private line that was updated by the current region to the LLC or any private cache of another core, the core sets its restart-forbidden bit. I/O operations, other system calls, and interrupts, which make a region *irrevocable* because their effects may be visible to the external world, also trigger setting the restart-forbidden bit.

When a core begins a region, it clears its restart-forbidden bit and saves a shadow copy of its current execution context, which is the same architecture state that the core normally saves on a context switch (i.e., register state, but not cache or memory contents). The core uses this shadow copy to initiate the re-execution of a region by restoring its execution context.

To handle a lazily detected conflict, Peacenik restarts a region involved in the conflict if it is eligible for restart (i.e., its restart-forbidden bit is cleared). To handle a pausing deadlock, the pausing controller selects an arbitrary core that has its restart-forbidden bit cleared, and signals the core to restart its region.

A core takes several steps to restart an eligible, ongoing region. To preserve atomicity, Peacenik needs to revert the updates performed by the restarting region. The core rolls back the region's writes by invalidating the private cache lines updated by the region, relying on a lower-level cache (or memory) to contain values corresponding to the region's start. Next the core restores its execution context (i.e., register file) from the shadow copy saved at region start, and starts executing again from the first instruction of the restarting region. The baseline fail-stop system may need to perform additional system-specific actions to handle the region restart, such as clearing access metadata used for conflict detection. Finally, the core prompts the pausing controller to reset the pause bits for any cores that are waiting for the restarting core.

Peacenik relies on the baseline fail-stop system having a mechanism for differentiating lines written by the restarting region from other dirty lines (i.e., lines written by the core's previous regions). Furthermore, the system must ensure that a dirty line is backed up in the cache hierarchy before being written by a new region; otherwise values that correspond to the region's start will be lost, thwarting region restart.

Peacenik cannot tolerate a lazy conflict or pausing deadlock if no involved core is eligible to restart. In such a case, Peacenik generates a consistency exception that terminates the program's execution. A feature of Peacenik is that it does not modify the baseline fail-stop architecture's consistency exception delivery mechanism. We assume the underlying architecture raises a dedicated per-core signal via a non-maskable interrupt to the executing core that, by default, is handled by operating system code that terminates program execution.

Region restart in ARC+Peacenik. In ARC+Peacenik, when a region restarts, the core clears its access information in the AIM, and clears access information for each read or written line in its private caches and then invalidates those lines. The core invalidates lines not only written by the restarting region, but also lines *read* by the region, in order to fetch up-to-date copies of them during re-execution of the region.

ARC already differentiates lines written by the restarting region from previously written lines. And it can generally invalidate a region's written lines without losing writes by previous regions—except in the presence of ARC's *deferred write-back* optimization, which defers writing back a region's written-to lines until another core accesses them [10]. Briefly, ARC+Peacenik ensures that such lines from a previous region are not overwritten by writes in the current region, by backing up the lines' values in the L2 when the lines are deferred during post-commit. ARC+Peacenik introduces a per-line *deferred-write bit* in the L2 to indicate such lines (Figure 2). If the L1 evicts a line for which the same line in the L2 has its deferred-write bit set, the L2 writes back its dirty values to the LLC and clears the deferred-write bit.

Region restart in CE+Peacenik. In CE+Peacenik, a core performs end-of-region actions as usual before restarting its region: clearing local access bits, sending end-of-region messages to other cores, and waiting for acknowledgments.

To support region restart, CE+Peacenik must differentiate lines written by the current region from previously written lines, and preserve values of data from region start time. Briefly, CE+Peacenik extends CE so that when a dirty line is written to for the first time by a region (i.e., no access bits have been set), the line is first written back to the next lowest level of the cache/memory hierarchy. Then invalidating a region's written-to lines restores the region start state.

4.4 Reducing Dirty Evictions to Shared Cache

To avoid dirty evictions to a shared cache, which render a core's region ineligible for restart, Peacenik modifies the *last-level private cache* (e.g., the L2 in Figure 2) replacement policy to prefer clean lines over dirty ones as eviction candidates. Peacenik adds this constraint to a baseline *pseudo-LRU* replacement policy, modeled after the one used in the Intel Core i7 memory hierarchy [30]. Baseline pseudo-LRU uses one *most recently used (MRU) bit* per line that the core sets each time it accesses the line, indicating that the line was recently accessed. When the MRU bits of all lines in a cache set become set, pseudo-LRU clears all of the set's MRU bits, except for the most recently set bit. Pseudo-LRU implements eviction by selecting an arbitrary line that has its MRU bit cleared.

To reduce the frequency of evictions of dirty lines, Peacenik modifies the replacement policy's behavior when clearing MRU bits: instead of clearing all MRU bits (except for the most recently set one), Peacenik sets each line's MRU bit equal to the value of the line's dirty bit. To select a line for eviction, Peacenik's modified pseudo-LRU evicts a line with its MRU bit cleared, which will be a clean line if any line is clean. If all of a set's MRU bits are set, then all of the lines in the set are dirty. If all of the lines in the set are dirty, Peacenik's replacement policy behaves like a normal pseudo-LRU policy, and chooses an arbitrary line to evict. The core sets its restart-forbidden bit if it evicts a dirty line in this case.

Due to inclusion, when a core evicts a line from L2, the corresponding line in L1 must be recalled and evicted. To avoid evicting a line that is clean in L2 but dirty in L1—which would make the region ineligible to restart—a core sets the dirty bit for a line in both L1 and L2 when the L1 line first becomes dirty in a region, making Peacenik's modified pseudo-LRU less likely to evict it.

Prior research on high-performance cache replacement policies shows that the effects of replacement decisions are significant only for the shared LLC, and modifying the replacement policy of the L2 cache is unlikely to have a significant impact on performance [33, 68].

ARC+Peacenik and CE+Peacenik use the modified L2 eviction policy exactly as described above.

4.5 Hardware Complexity

The hardware components added by Peacenik on top of the underlying systems that provide fail-stop consistency have low complexity and do not contribute significantly to area or static power. Per-core pause and restart-forbidden bits contribute negligibly to complexity. Peacenik adds a per-line deferred-write bit to the L2 cache, which amounts to only 0.1% of the L2 cache. The wait list in the pausing controller contains one entry per core, and each entry identifies a wait-for core. With c cores, the overhead of the wait list is $c \times log_2(c)$ bits, which is just 20 bytes of storage for 32 cores. The logic to check for pausing is not on the critical path of cache data or tag accesses and will not impact cycle time. This checking logic relies on simple table lookups and single-bit comparators (for checking pause and restart-forbidden bits) and does not contribute significantly to complexity. Peacenik's additions on top of existing support for fail-stop consistency increase complexity only negligibly and are not likely to significantly impact power, area, or performance.

4.6 Thread Migration and Context Switches

The discussion so far has assumed that each core executes a single, pinned thread. Here, we discuss how Peacenik supports thread switching and migration.

Peacenik assumes that the underlying fail-stop system (e.g., ARC or CE) already supports migrating and switching threads. Peacenik additionally must virtualize per-thread pausing and restart state. Each thread must include a pause bit and a wait-for entry in its execution context that is saved with the rest of the context through switches and migration, and used to set a core's pause bit and waiting-for status when the thread is scheduled or migrated to the core. A paused core can switch threads; it is likely advantageous to switch to a non-paused thread.

Although each thread could add a restart-forbidden bit to its execution context, it seems difficult to support restart for a migrated or switched thread. A straightforward option is to simply set a core's restart-forbidden bit whenever it switches to a different thread.

It is straightforward to extend Peacenik to support simultaneous multithreading (SMT), by adding pause and restart-forbidden bits and a wait-for entry in the wait list for each *logical* core, and applying Peacenik's mechanisms to logical cores.

Pausing interacts with I/O and interrupts similarly to how pausing responds to thread context switches. We argue that a system that provides fail-stop semantics should avoid detecting conflicts when executing privileged code such as system calls.

5 Evaluation

This section evaluates Peacenik's ability to avoid consistency exceptions and the resulting impact on performance.

5.1 Evaluation Methodology

We implemented the two Peacenik prototypes, ARC+Peacenik and CE+Peacenik, by extending simulators for ARC and CE [10, 41] that we implemented for the ARC paper [10].[4] We have made our ARC+Peacenik and CE+Peacenik implementations publicly available.[5]

The simulators consume a trace of instructions generated by a Pin-based front end [42] and model a simple, in-order core architecture with 8–32 cores.[6] Table 2 shows the main parameters used for the modeling. We report execution time as the maximum cycles of any core. The ARC and ARC+Peacenik simulators model a *non-inclusive* LLC, while the CE and CE+Peacenik simulators model an *inclusive* LLC to support an inclusive directory cache embedded in the LLC [62].

Workloads. Our experiments run the PARSEC 3.0 benchmarks [7] with simsmall inputs, except we use simmedium for swaptions because its simsmall workload does not support >16 threads. We exclude facesim and raytrace, which fail to finish executing with the simulators, and freqmine,

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

[5] https://github.com/PLaSSticity/peacenik-simulators-asplos20

[6] While trace-based simulation and in-order core modeling would hide weak memory model effects, they should not significantly affect the prevalence of conflicts between synchronization-free regions.

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

Table 2. Architectural parameters used for simulation.

which uses OpenMP instead of pthreads. The simulators only compute cycles for each PARSEC program's "region of interest" (ROI), which includes the whole parallel phase of each program.

The experiments also execute two real server programs: Apache HTTP Server 2.4.23 (httpd) and MySQL Server 5.7.16 (mysqld) [1, 2]. We configure both programs to create a single child process with n worker threads, where n is the number of cores in the simulated architecture. We launch n client processes that repeatedly and concurrently perform simple requests using a methodology documented by prior work [70].[7] In our experiments, client processes executing natively send 128K HTTP requests (httpd) or 256 SQL queries (mysqld) to the server, distributed evenly over the client processes.

We compiled the PARSEC benchmarks without optimizations (i.e., gcc -O0) and mysqld with -O1, for two reasons. First, our Pintool front end identifies synchronization operations according to specific pthreads function names. At higher optimization levels, these names can be optimized away, leading to missing synchronization boundaries and thus false conflicts. Second, with higher optimization levels our Pintool is unable to consistently get the source locations involved in conflicts, which we need to report, analyze, and understand the conflicts reported in Section 5.5. We performed a separate evaluation and found no significant difference between the unoptimized and optimized experiments in terms of the relative performance differences between configurations. We compiled httpd at its default optimization level (-O2) because the default optimizations did not lead to missing synchronization functions or inconsistent source locations in our experiments.

Handling synchronization. All simulators identify calls to pthreads functions (and mysqld's PolicyMutex.enter() and PolicyMutex.exit()) as lock operations. At a lock operation, ARC and ARC+Peacenik execute a distributed queue-based locking protocol [10, 63] and end the current synchronization-free region (SFR) by performing the region commit protocol, while CE and CE+Peacenik end the current SFR by performing end-of-region actions.

ARC treats non-pthreads atomic instructions (e.g., inlined assembly or C++ atomic accesses [14]) as synchronization but not region boundaries. To implement that behavior, the ARC and ARC+Peacenik simulators treat non-pthreads atomic instructions (i.e., instructions with the x86 LOCK prefix and fence instructions; the evaluated programs do not use C++ atomic variables; atomic instructions in the experiments originate from inlined assembly in the source code) as lock operations but *not* region boundaries. CE and CE+Peacenik treat non-pthreads atomic instructions as regular memory accesses but do not detect conflicts on the instructions. As a result, the simulators do not detect conflicts on atomic instructions, but they may detect false conflicts on accesses to *other* variables happens-before-ordered by the atomic instructions. The simulators detected such false conflicts in several low-level library functions: malloc, start_thread, dlinfo, and seekoff. We modified the simulators to ignore these conflicts. A deployed system using ARC and CE should identify such atomic-instruction-ordered accesses in library functions as atomic.

Modeling of pausing, region restarts, consistency exceptions, and reboots. The simulated cost of Peacenik pausing a core C is the cycles executed by the wait-for core C_w while C is paused, transitively including any cycles that core C_w spent in a paused state while waiting for other cores. To achieve realistic thread interleavings with pausing, the simulator backend "pauses" the application thread by signaling back to the Pintool front end, which stops executing the thread until signaled to continue by the simulator backend.

The simulators model the cost of region restart by re-executing the region's instructions in the simulator, but not in the actual application. To support re-execution, the simulator buffers each core's region's instructions until commit. Our evaluation faithfully counts the performance cost no matter how many times a region restarts.

For PARSEC, we assume that ARC, CE, and Peacenik handle a consistency exception by *rebooting* the application, i.e., restarting it from the beginning. The simulators model the cost of rebooting by adding the total cycles incurred so far (not including prior reboots) to the overall computed

[7] We reuse methodology from https://github.com/jieyu/concurrency-bugs, but use more recent versions of httpd and mysqld, which still have data races that manifest as region conflicts.

Config	Action taken on detected... eager conflict	lazy conflict*	pausing deadlock	L2 replacement policy
ARC or CE	Consistency exception	Consistency exception	N/A	Default pseudo-LRU
PNp	Pause core	Consistency exception	Consistency exception	Default pseudo-LRU
PNpr	Pause core	Restart region†	Restart region†	Default pseudo-LRU
PN	Pause core	Restart region†	Restart region†	Modified pseudo-LRU

Table 3. Behaviors of each simulated configuration, characterized by actions taken on eagerly and lazily detected conflicts and on pausing deadlocks, and whether using the modified pseudo-LRU replacement policy in the L2 cache. ∗ CE and CE+Peacenik do *not* detect any conflicts lazily. † If a region is ineligible to restart, Peacenik generates a consistency exception.

costs. If a server program (httpd or mysqld) encounters a consistency exception while processing a request, we assume the server can safely retry just the exceptional core's current request. The simulators model retried requests by adding the request's cycles so far (not including prior retries) to the core's total costs. If a server program generates a consistency exception while *not* processing a request, the simulators model a reboot by adding only the cycles needed to start the server, not to repeat any requests, to the overall computed costs.

5.2 Peacenik's Run-Time Impact

This section evaluates Peacenik's ability to avoid consistency exceptions and thus reduce execution costs compared with ARC and CE. To break down the benefit of Peacenik's mechanisms, we use the following configurations, which extend the baseline system (ARC or CE):

- *PNp* is a partial Peacenik configuration that *pauses* at eager conflicts, but does *not* restart regions.
- *PNpr* extends PNp to *restart* eligible regions that encounter a lazy conflict (if applicable) or pausing deadlock.
- *PN* is the full, default Peacenik configuration. It extends PNpr by modifying pseudo-LRU to avoid dirty L2 evictions.

The PN configuration faithfully performs Peacenik's modified pseudo-LRU algorithm and thus incurs any performance effects resulting from potentially suboptimal eviction decisions. Other Peacenik and non-Peacenik configurations faithfully perform unmodified pseudo-LRU.

Table 3 summarizes the behavior of each configuration.

Table 4 and Figure 3 present the main results of the evaluation. They show results for the four programs for which ARC and CE detect conflicts for 32 cores: canneal, streamcluster, httpd, and mysqld. For each program, all ARC and ARC+Peacenik configurations process the same execution trace generated by the Pintool front end; similarly, CE and CE+Peacenik configurations process the same trace. We run ARC and CE experiments separately because ARC and CE use different conflict detection mechanisms, and hence their pausing behavior differs and leads to different traces.

In Table 4, the *Ordinary regions (cycles)* column is each program's total executed SFRs and executed cycles, *excluding* extra regions and cycles executed due to pausing and restarting. The *Regions w/...* columns count regions with at least one eagerly detected conflict (EC), lazily detected conflict (LC), and pausing deadlock (PDL), respectively. Peacenik's pausing inherently avoids eagerly detected conflicts, so *EC* actually represents *conflicts avoided by pausing* for all Peacenik configurations. The *Region restarts* column counts eligible regions restarted by Peacenik. The *Retried requests (avg. cycles)* column counts retried requests and reports average cycle cost of each retried request (applicable to server programs only). *App reboots (avg. cycles)* counts application reboots and shows average cycle cost of each reboot.

Figure 3 shows run time for the same configurations and programs as Table 4. The graphs break down run time into multiple components. All configurations incur costs due to *ordinary execution* and *handling exceptions*. Each program's bars are normalized to the *ordinary execution* cost of the baseline (ARC or CE alone). Peacenik includes costs for *pausing* and *restarting regions*, although costs are small in the experiments and not visible in most cases.

PARSEC programs. The simulators report hundreds of conflicts in canneal and streamcluster. ARC and CE generate consistency exceptions at these conflicts and incur high slowdowns (76–240× for canneal and 240–670× for streamcluster) from substantial re-execution of the program from the start. For example, as Table 4 shows, ARC's consistency exceptions in canneal incur 20 million cycles per reboot in an execution of only 47 million cycles. ARC and CE detect different numbers of conflicts (and in general, CE detects more conflicts than ARC) because (1) CE detects all conflicts eagerly and (2) ARC and CE detect conflicts at different points (which explains why ARC detects a different number of conflicts than CE for canneal, despite detecting no lazy conflicts).

In contrast, Peacenik avoids much or even all of the cost of consistency exceptions. For canneal, ARC+Peacenik pauses at all 189 eagerly detected conflicts (more than the 173 conflicts reported by ARC because pausing leads to different timing and thus different region conflicts), instead introducing

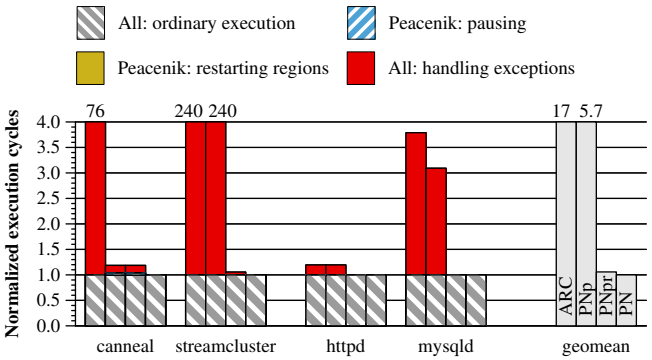
(a) ARC and ARC+Peacenik configurations

App	Config	Ordinary regions (cycles)	Regions w/ EC	Regions w/ LC	Regions w/ PDL	Region restarts	Retried requests (avg. cycles)	App reboots (avg. cycles)
canneal	ARC		173	0	–	–	–	173 (20M)
	PNp	1,280	189	0	2	–	–	2 (3.4M)
	PNpr	(47M)	189	0	2	0	–	2 (3.4M)
	PN		0	0	0	0	–	0
streamcluster	ARC		0	566	–	–	–	566 (160M)
	PNp	368,550	0	566	0	–	–	566 (160M)
	PNpr	(380M)	0	566	0	565	–	1 (20M)
	PN		0	566	0	566	–	0
httpd	ARC		3	3	–	–	2 (200K)	4 (26M)
	PNp	1,177,315	5	4	2	–	2 (210K)	4 (26M)
	PNpr	(530M)	7	4	3	7	0	0
	PN		7	4	3	7	0	0
mysqld	ARC		2	11	–	–	9 (840K)	4 (790M)
	PNp	905,356	2	12	0	–	9 (840K)	3 (790M)
	PNpr	(1,100M)	0	11	0	11	0	0
	PN		0	11	0	11	0	0

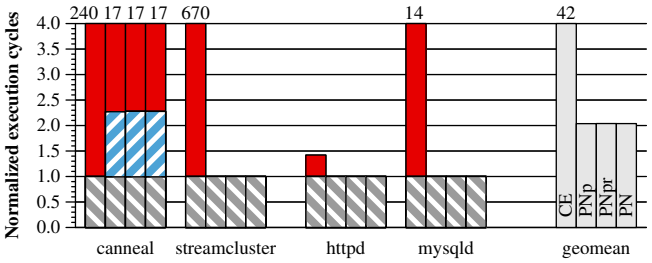
(b) CE and CE+Peacenik configurations

App	Config	Ordinary regions (cycles)	Regions w/ EC	Regions w/ PDL	Region restarts	Retried requests (avg. cycles)	App reboots (avg. cycles)
canneal	CE		516	–	–	–	516 (20M)
	PNp	1,216	493	17	–	–	17 (39M)
	PNpr	(44M)	493	17	0	–	17 (39M)
	PN		492	17	0	–	17 (39M)
streamcluster	CE		1,672	–	–	–	1,672 (140M)
	PNp	368,563	790	0	–	–	0
	PNpr	(360M)	790	0	0	–	0
	PN		790	0	0	–	0
httpd	CE		8	–	–	0	8 (26M)
	PNp	1,172,175	8	0	–	0	0
	PNpr	(490M)	8	0	0	0	0
	PN		8	0	0	0	0
mysqld	CE		19	–	–	0	19 (750M)
	PNp	967,908	10	0	–	0	0
	PNpr	(1100M)	10	0	0	0	0
	PN		10	0	0	0	0

Table 4. Run-time characteristics of programs for which ARC and CE detect conflicts, on 32 cores. A blank entry (–) indicates an inapplicable case for the configuration or program. The table reports cycles in thousands (K) or millions (M), rounded to two significant figures. EC: eager conflicts, LC: lazy conflicts, PDL: pausing deadlocks



(a) ARC and ARC+Peacenik configurations



(b) CE and CE+Peacenik configurations

Figure 3. Execution time for the baseline systems ARC and CE, and the Peacenik configurations based on them, on 32 cores, normalized to each baseline system's *ordinary execution* cost. Overflow numbers are rounded to two significant figures.

just 2 pausing deadlocks due to cyclic pausing dependencies—both of which occur early in execution, leading to low reboot costs. The pausing deadlocks occur in regions that are ineligible to restart, and PNpr does not provide more benefit than PNp. PN benefits canneal indirectly by avoiding evictions of written-to lines that lead to canneal's write–write conflict (Section 5.5). As a result, the conflicting regions complete without conflict. CE+Peacenik pauses at all 493 conflicts (less than the 516 conflicts reported by CE for the same reason as explained above for ARC+Peacenik) but introduces 17 pausing deadlocks. All of the pausing deadlocks occur in regions that are ineligible to restart, and region restart in PNpr and PN cannot avoid them. Modified pseudo-LRU does not provide as much benefit as in ARC+Peacenik since the regions are ineligible for restart due to remote cache requests of their written-to lines rather than evictions. But PN's modified pseudo-LRU enables detecting 1 fewer conflict than other CE+Peacenik configurations because the pausing controller identifies the wait-for core for a paused core by checking *in order* the conflicting line's local bits in each core's private caches and the global table if the line was evicted by its core (Section 4.2). When multiple cores concurrently write to the same location and cause a conflict, different wait-for cores may be identified under different replacement policies depending on where the conflicting line resides (private caches or the LLC), leading to different timing and thus different region conflicts. CE+Peacenik incurs a slowdown of 17 ×, much less than CE's 240 ×.

For streamcluster, all consistency exceptions generated by ARC are for lazily detected conflicts that pausing cannot avoid and thus PNp does not reduce costs. PNpr avoids all but one consistency exception because all regions except one are

eligible to restart, which substantially improves run time. By modifying pseudo-LRU, PN enables one region to restart that would otherwise be ineligible to restart, avoiding consistency exceptions entirely. CE+Peacenik detects all conflicts eagerly and thus pausing avoids all consistency exceptions, without incurring any pausing deadlocks.

Server programs. For httpd, ARC generates 6 consistency exceptions: 2 occur when processing a request, allowing retrying the request; the other 4 occur outside of request processing, requiring a server reboot. Reboots incur a relatively low cost: booting takes only 5% of httpd's ordinary cycles. PNp detects 9 conflicts, again a few more than ARC due to pausing's impact on region timing. PNp does not reduce consistency exceptions for httpd because PNp has extra conflicts and pausing causes 2 pausing deadlocks. For the PNpr and PN configurations, all regions with lazily detected conflicts or pausing deadlocks are eligible for restart, and ARC+Peacenik avoids all consistency exceptions. CE detects 8 eager conflicts; pausing avoids all conflicts without any pausing deadlocks.

For mysqld, ARC generates consistency exceptions due to 13 regions with conflicts, 9 of which occur in requests. Booting mysqld is costly (790M cycles), and ARC incurs a 3–4× slowdown to reboot 4 times. Pausing alone (i.e., PNp) avoids 2 consistency exceptions from eagerly detected conflicts, but pausing-related timing differences lead to extra lazy conflicts, resulting in modest net reductions in run time. Peacenik configurations that support restarting regions (PNpr and PN) detect no conflicts eagerly, as a result of execution differences caused by restarting regions and modifying ARC's deferred write-back optimization (Section 4.3). The support for restarting regions helps avoid all consistency exceptions by restarting eligible regions with lazily detected conflicts, whether using modified pseudo-LRU or not. CE generates 19 consistency exceptions, all of which occur outside of request processing and lead to server reboots. Pausing leads to no pausing deadlocks and thus avoids all consistency exceptions with minimal cost.

In summary, the results in this section show that when applied to a system that provides fail-stop semantics by detecting conflicts both eagerly and lazily, all of Peacenik's components—pausing, region restart, and modified pseudo-LRU—are collectively important for avoiding consistency exceptions, together eliminating nearly all of the performance impact of fail-stop semantics for data races. Region restart and modified pseudo-LRU are particularly important for a system like ARC that detects conflicts lazily, while an eager-conflicts-only system like CE gets by with pausing alone. The results thus show the effectiveness and generality of Peacenik applied to systems that provide fail-stop behavior for data races.

5.3 Scalability

Figure 4 evaluates Peacenik on 8-, 16-, and 32-core systems. The results include a *weak memory model* (WMM) configuration, which models a typical multiprocessor architecture that *ignores* data race errors, to compare with the current state of practice. We implemented the WMM configuration by disabling CE's components in the CE simulator. WMM does not detect conflicts or generate consistency exceptions. As in Section 5.2, the ARC (Figure 4(a)) and CE (Figure 4(b)) results are each from a separate set of runs.

We note that for vips with 8 and 16 cores, CE detects conflicts that are not detected with 32 cores because of different timing and data contention patterns. Conversely, with 8 cores CE misses all conflicts detected in mysqld with 16 and 32 cores.

Overall, the data show that across core counts, Peacenik not only avoids nearly all consistency exceptions in our experiments, eliminating performance degradation, but it negligibly impacts conflict-free programs relative to the underlying fail-stop system (ARC or CE).

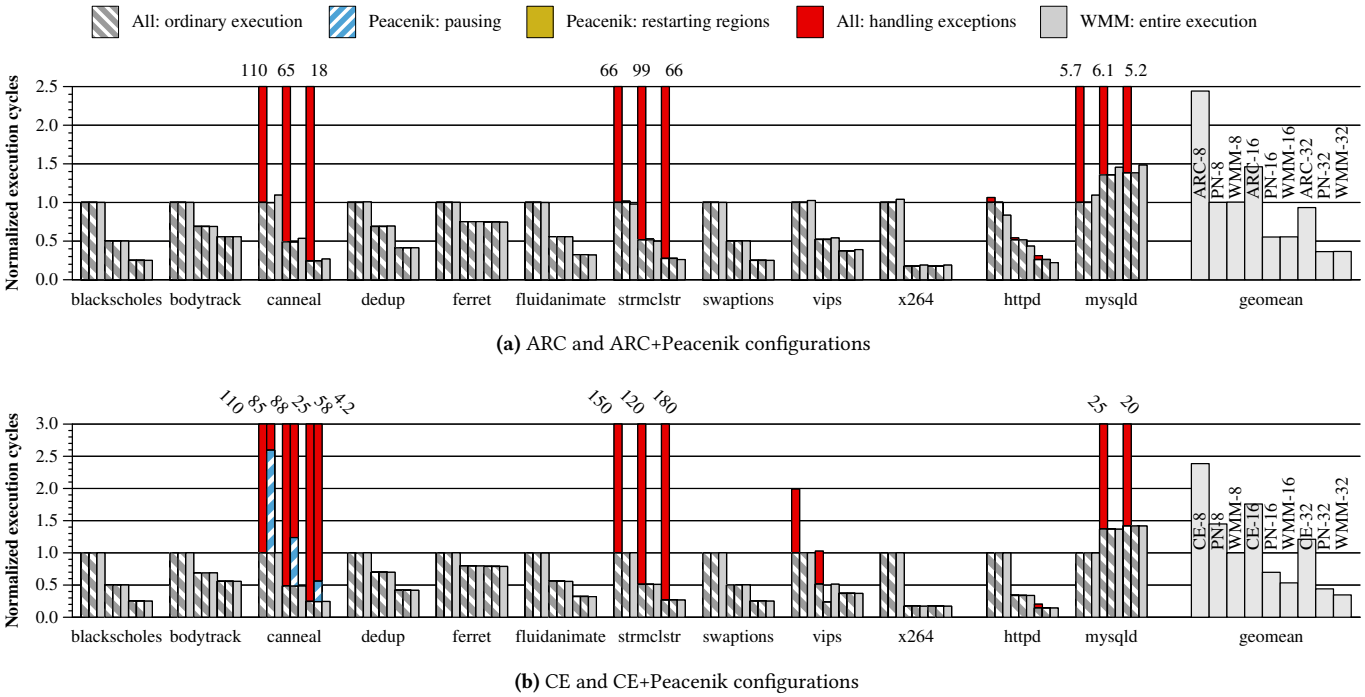
5.4 Idealized Checkpointing as a Rebooting Alternative

The evaluation so far has assumed that ARC and CE handle a consistency exception by rebooting the program—a high price to pay. Instead, programs executing under fail-stop memory consistency could periodically record a *process checkpoint*, in order to return the execution's state to the most recent checkpoint if a core encounters a failure.

Here we evaluate a highly *idealized* checkpointing implementation that incurs *no cost* for taking checkpoints. The actual cost of creating and maintaining process checkpoints would increase costs, making Peacenik by contrast look even more favorable. We emphasize that checkpointing is distinct from Peacenik's support for best-effort region restart: checkpointing must record a process's memory contents, whereas region restart only requires recording a shadow copy of the execution context at region start (Section 4.3).

Figure 5 shows the run time of unmodified ARC and ARC+Peacenik (for simplicity this experiment excludes CE and CE+Peacenik), compared to ARC configurations that perform idealized checkpointing (labeled *ARC-cp{1,...,1000}* and *ARC-noreboot*), running on 32 cores.

For canneal and streamcluster, the system checkpoints at global synchronization barriers that demarcate the programs' outer-loop iterations. For each ARC-cp*k* configuration, the system checkpoints every *k* iterations (e.g., ARC-cp10 checkpoints every 10 iterations). At a consistency exception, the simulator adds run time assuming re-execution from the last checkpoint, instead of a full reboot. These ARC-cp*k* configurations improve run time compared with ARC: 6–27× for canneal and 3–170× for streamcluster. However, even checkpointing at every barrier (i.e., ARC-cp1) slows both



(a) ARC and ARC+Peacenik configurations

(b) CE and CE+Peacenik configurations

Figure 4. Execution time for ARC/CE, full Peacenik (PN), and a weak memory model (WMM) for 8, 16, and 32 cores, normalized to the *ordinary execution* cost of ARC/CE with 8 cores, using the same methodology as for Figure 3. ARC-32, CE-32, and PN-32 are the same as Figure 3's ARC, CE, and PN.

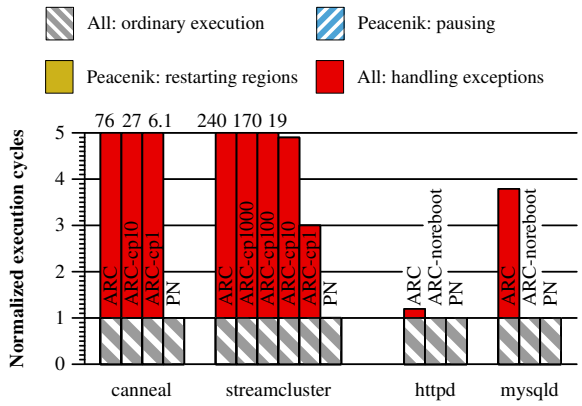


Figure 5. Execution time for ARC configurations with different process checkpointing rates, compared with PN without checkpointing, for 32 cores. Results are normalized to the *ordinary execution* cost of ARC.

programs by several times. In contrast, Peacenik eliminates all costs due to re-execution, by avoiding consistency exceptions altogether.

For httpd and mysqld, the *ARC-noreboot* configuration takes a checkpoint immediately after booting, to avoid the cost of rebooting. By avoiding rebooting through checkpointing, ARC-noreboot avoids virtually all of the cost of ARC's consistency exceptions, although a realistic checkpointing scheme would incur costs from checkpointing.

Overall, checkpointing reduces the impact of consistency exceptions, but even idealized checkpointing cannot outperform Peacenik in our evaluation. Peacenik and process checkpointing are complementary and could be deployed together.

5.5 Analysis of Detected Conflicts

This section analyzes the conflicts detected in the experiments (by ARC only, for simplicity), which we have confirmed as true data races using existing race detection tools. These conflicts are directly representative of data races that exist in mature legacy code, despite being erroneous under the C/C++ memory model [3, 14], and will lead to consistency exceptions when deployed on an SFRSx system.

We extended the ARC simulator to report the source locations of each conflict, including files and line numbers, by tracking the instruction that last accessed each byte in each cache line. Table 5 shows conflicting sites reported by ARC with 32 cores. Table 5 reports more dynamic conflicts than Table 4 because ARC may report multiple conflicting sites for a conflicting region.

We confirmed that the conflicts correspond to true data races using *ThreadSanitizer* (TSan) [57] and our own Pintool implementation of *collision analysis*, which pauses threads

	Variable name	Conflicting sites	Dyn	✓?
canneal	p	AtomicPtr.h:296⇋296	202	TS
stream-cluster	Point.*	streamcluster.cpp:727⇋729, etc.	475	CA
	gl_cost_of_opening_x	streamcluster.cpp:1120⇋1147	326	TS
httpd	cache_element	util_time.c:76⇋135	25	CA
	request_time_cache	mod_log_config.c:740⇋766	15	CA
	requests_this_child	worker.c:994⇋994	4	[59]
mysqld	curr (a.k.a. node)	lf_hash.c:99⇋267, etc.	44	TS*
	PFS_single_stat.*	pfs_stat.h:102⇋102, etc.	6	TS
	lock	sync0rw.ic:394⇋394, etc.	5	TS†
	buf_pool	buf0buf.cc:4089⇋4089	2	TS
	ut_rnd_ulint_counter	ut0rnd.ic:92⇋92	2	TS
	pfs	pfs.cc:2405⇋actor.cc:313, etc.	2	–
	digest_stat	pfs.cc:5525⇋5525	1	CA

Table 5. Conflicts detected by ARC in the evaluated programs, on 32 cores. *Variable name* shows the source name of each variable involved in a conflict; Point.* and PFS_single_stat.* indicate structs with multiple conflicting fields. *Conflicting sites* shows the static source file(s) and line number(s) in one conflict; the table abbreviates pfs_setup_actor.cc as actor.cc. *Dyn* is dynamic occurrences of conflicts for each variable. The last column shows if the conflict was confirmed by TSan *(TS)*, collision analysis *(CA)*, or prior work [59], or is unconfirmed *(–)*. The text explains *TS** and *TS†*.

to make accesses conflict [8, 22, 54]. We note that the conflict in canneal involves writes to a variable that is usually accessed atomically, but is accessed with a regular memory write at line 296 of AtomicPtr.h. Confirming that mysqld's if_hash.c conflict (*TS**) is a true data race required some manual inspection, due to TSan and ARC's different handling of a conflict between atomic and non-atomic accesses. TSan and collision analysis have confirmed some but not all of the source locations reported as conflicting for the lock variable in mysqld (*TS†*). We have not yet confirmed pfs.cc ⇋ actor.cc as a true data race (–).

6 Conclusion

Peacenik provides best-effort avoidance of failures on top of systems that provide fail-stop memory consistency. A simulation-based evaluation shows that Peacenik eliminates nearly all of the costs of fail-stop consistency while providing strong behavior guarantees for all programs. Peacenik shows how production systems can get the benefits of fail-stop consistency while minimizing the costs of failures.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This work is supported by the National Science Foundation under Grants CAREER-1253703, CCF-1421612, and XPS-1629126 and by compute credits awarded by the Google Cloud Platform (GCP) Education Programs Team.

References

[1] MySQL, 2017. https://www.mysql.com/products/community/.
[2] The Apache HTTP Server Project, 2017. https://httpd.apache.org/.
[3] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.
[4] Sarita V. Adve and Mark D. Hill. Weak Ordering—A New Definition. In *ISCA*, pages 2–14, 1990.
[5] Wonsun Ahn, Shanxiang Qi, Marios Nicolaides, Josep Torrellas, Jae-Woo Lee, Xing Fang, and David Wong. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO*, pages 133–144, 2009.
[6] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *HPCA*, pages 316–327, 2005.
[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, pages 72–81, 2008.
[8] Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, and Benjamin P. Wood. Lightweight Data Race Detection for Production Runs. In *CC*, pages 11–21, 2017.
[9] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.
[10] Swarnendu Biswas, Rui Zhang, Michael D. Bond, and Brandon Lucia. Rethinking Support for Region Conflict Exceptions. In *IPDPS*, pages 1095–1106, 2019.
[11] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *ISCA*, pages 24–34, 2007.
[12] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA*, pages 127–138, 2008.
[13] Hans-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.
[14] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.
[15] Hans-J. Boehm and Brian Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*, pages 7:1–7:6, 2014.
[16] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, pages 278–289, 2007.
[17] Yuxi Chen, Shu Wang, Shan Lu, and Karthikeyan Sankaralingam. Applying Hardware Transactional Memory for Concurrency-Bug Failure Recovery in Production Runs. In *USENIX*, pages 837–850, 2018.
[18] Christian DeLozier, Ariel Eizenberg, Brandon Lucia, and Joseph Devietti. SOFRITAS: Serializable Ordering-Free Regions for Increasing Thread Atomicity Scalably. In *ASPLOS*, pages 286–300, 2018.
[19] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ISCA*, pages 201–212, 2012.
[20] Yuelu Duan, David Koufaty, and Josep Torrellas. SCsafe: Logging sequential consistency violations continuously and precisely. In *HPCA*, pages 249–260, 2016.
[21] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, 2007.
[22] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, pages 1–16, 2010.
[23] Mahdi Eslamimehr and Jens Palsberg. Race Directed Scheduling of Concurrent Programs. In *PPoPP*, pages 301–314, 2014.
[24] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[25] Kourosh Gharachorloo and Phillip B. Gibbons. Detecting Violations of Sequential Consistency. In *SPAA*, pages 316–326, 1991.

[26] Chris Gniady and Babak Falsafi. Speculative Sequential Consistency with Little Custom Storage. In *PACT*, pages 179–188, Washington, DC, USA, 2002.

[27] Patrice Godefroid and Nachi Nagappan. Concurrency at Microsoft – An Exploratory Survey. In *(EC)*^{2}, 2008.

[28] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, pages 102–113, 2004.

[29] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[30] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

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

[32] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and T. N. Vijaykumar. Wait-n-GoTM: Improving HTM Performance by Serializing Cyclic Dependencies. In *ASPLOS*, pages 521–534, 2013.

[33] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *ISCA*, pages 60–71, 2010.

[34] Baris Kasikci, Cristian Zamfir, and George Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, pages 185–198, 2012.

[35] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: Crowdsourced Data Race Detection. In *SOSP*, pages 406–422, 2013.

[36] Baris Kasikci, Cristian Zamfir, and George Candea. Automated Classification of Data Races Under Both Strong and Weak Memory Models. *TOPLAS*, 37(3):8:1–8:44, May 2015.

[37] Nancy G. Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[38] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient Sequential Consistency Using Conditional Fences. In *PACT*, pages 295–306, 2010.

[39] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, pages 273–286, 2012.

[40] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[41] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, pages 190–200, 2005.

[43] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[44] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[45] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, pages 199–210, 2011.

[46] Don P. Mitchell and Michael J. Merritt. A Distributed Algorithm for Deadlock Detection and Resolution. In *PODC*, pages 282–284, 1984.

[47] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

[48] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions, 2012. http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[49] Yuanfeng Peng, Benjamin P. Wood, and Joseph Devietti. PARSNIP: Performant Architecture for Race Safety with No Impact on Precision. In *MICRO*, pages 490–502, 2017.

[50] Xuehai Qian, Benjamin Sahelices, and Josep Torrellas. BulkSMT: Designing SMT Processors for Atomic-block Execution. In *HPCA*, pages 1–12, 2012.

[51] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware Transactional Memory for Increased Concurrency. In *MICRO*, pages 246–257, 2008.

[52] Parthasarathy Ranganathan, Vijay Pai, and Sarita Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *SPAA*, pages 199–210, 1997.

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

[54] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, pages 11–21, 2008.

[55] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[56] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. Legato: End-to-End Bounded Region Serializability Using Commodity Hardware Transactional Memory. In *CGO*, pages 1–13, 2017.

[57] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic Race Detection with LLVM Compiler. In *RV*, pages 110–114, 2012.

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

[59] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. RACEZ: A Lightweight and Non-Invasive Race Detection Tool for Production Applications. In *ICSE*, pages 401–410, 2011.

[60] Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, pages 53–66, 2011.

[61] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-End Sequential Consistency. In *ISCA*, pages 524–535, 2012.

[62] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[63] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. In *ASPLOS*, pages 13–26, 2013.

[64] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, pages 2–13, 2005.

[65] U.S.–Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.

[66] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *PACT*, pages 127–136, 2012.

[67] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*, pages 671–686, 2014.

[68] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. In *MICRO*, pages 430–441, 2011.

[69] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19:1–19:11, 2013.

[70] Jie Yu and Satish Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *ISCA*, pages 325–336, 2009.

[71] Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. Avoiding Consistency Exceptions Under Strong Memory Models. In *ISMM*, pages 115–127, 2017.

[72] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. ConAir: Featherweight Concurrency Bug Recovery via Single-threaded Idempotent Execution. In *ASPLOS*, pages 113–126, 2013.

A Artifact Appendix

A.1 Abstract

This artifact provides a VirtualBox image that includes source code and x86-64 binaries of our simulators and evaluated PARSEC 3.0 benchmarks and real server programs.

We also provide scripts to build and run the simulators with the benchmarks and server programs and to generate the performance graphs in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Simulated Peacenik architecture mechanisms based on the ARC and CE architectures.
- **Program:** The ARC and ARC+Peacenik, CE and CE+ Peacenik, and WMM simulators; a Pin-based front end; a framework to drive experiments; and 10 PARSEC benchmarks and two server programs.
- **Data set:** Test and simsmall workloads for the PARSEC benchmarks and custom workloads for the server programs as described in Section 5.1.
- **Run-time environment:** VirtualBox 5.2.34 or above.
- **Hardware:** Memory ≥ 16 GB and cores ≥ 4 for test workloads; memory ≥ 104 GB and cores ≥ 16 to reproduce results in the paper.
- **Output:** Execution cycles graphs and reports of detected region conflicts.
- **How much time is needed to complete experiments (approximately)?:** Several weeks if all experiments are run sequentially. Using multiple VMs to run the experiments in parallel is highly recommended.
- **Publicly available?:** Yes.
- **Archived?:** https://doi.org/10.5281/zenodo.3603351

A.3 Description

A.3.1 How delivered:

VirtualBox image is publicly available on Zenodo: https://doi.org/10.5281/zenodo.3603351.

A.3.2 Hardware dependencies:

In order to reproduce the data for the graphs in the paper, the hardware should be able to host one or more VMs, each with ≥ 16 cores and 104 GB memory.

A.3.3 Software dependencies:

VirtualBox 5.2.34 or above on Ubuntu 18.04 or RHEL 7 is recommended.

A.4 Installation

The VirtualBox image can be imported into VMs with either the VirtualBox GUI or from the command line. Make sure your VirtualBox kernel module is correctly loaded using the following command for Ubuntu:

```
$ systemctl status virtualbox
```

or the following command for RHEL:

```
$ systemctl status vboxdrv
```

For importing using GUI, refer to the official manual at https://www.virtualbox.org/manual/ch01.html#ovf. Make sure to allocate sufficient memory and CPUs as listed in Section A.3.2. Both the username and password are "asplos20" when needed after successfully importing the image.

In order to import the image from the command line to create a VM with sufficient hardware resources (e.g., 32 cores and 128 GB memory) to reproduce the results in the paper, use the following commands.

```
$ VBoxManage import asplos20-ae-img.ova --vsys 0 --cpus 32 --memory 131072 --vmname asplos20-ae-vm
$ VBoxManage startvm asplos20-ae-vm --type headless
$ ssh asplos20@localhost -p 2200
```

Port 2200 of the host has been forwarded to port 22 of the VM/guest to enable ssh access. If the host's port 2200 has been assigned to another process (in which case the above command will fail), you may want to change the port forwarding. For example, to change to port 2222 instead, use the following commands.

(If VM is off) `$ VBoxManage modifyvm "asplos20-ae-vm" --natpf1 "guestssh,tcp,,2222,,22"`

(If VM is on) `$ VBoxManage controlvm "asplos20-ae-vm" natpf1 "guestssh,tcp,,2222,,22"`

Type the password "asplos20" when prompted.

A.5 Experiment workflow

In the VM, use the scripts to run simulation as follows.

```
$ cd /home/asplos20/scripts
```

Use the test.sh script to run simulation on test workload as follows.

```
$ ./test.sh output_dir
```

Go to /home/asplos20/exp-products/output_dir for the resulting data tables and graphs.

Use the exp-figX.sh scripts to run simulation to generate data for Figure X in the paper. For example, to reproduce Figure 3(a),

```
$ nohup ./exp-fig3a.sh output_dir > fig3a.out &
```

A.6 Evaluation and expected result

After running the experiments, expect the following.

1) Graphs similar to those in the paper. All graphs are generated in /home/asplos20/exp-products in the VM.

2) Reports of detected conflicts as reported in Table 5, if ARC configurations are used. Reports are output directly to stdout.

A.7 Experiment customization

Users can change the scripts to customize experiments.

A.8 Notes

See /home/asplos20/scripts/README in the VM for more information about using the scripts, viewing results, and customizing experiments.

Check out source code of our experiment framework and simulators at the following locations in the VM ($HOME= /home/asplos20).

- **Experiment framework:** $HOME/viser-exp
- **Pin-based front end:** $HOME/intel-pintool/source/tools/ViserST
- **ARC and ARC+Peacenik simulators:** $HOME/visersim-backend
- **CE, CE+Peacenik, and WMM simulators:** $HOME/mesisim-backend

A.9 Methodology

Submission, reviewing, and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging