Arm Weak Memory Consistency on Apple Silicon: What Is It Good For?

Yossi Khayet
Tel Aviv University
Tel Aviv, Israel
johe97@gmail.com

Adam Morrison
Tel Aviv University
Tel Aviv, Israel
mad@cs.tau.ac.il

Abstract

Weak memory models such as the Arm model are perceived as enabling higher performance than strong models such as TSO. We critically test this perception on Apple silicon CPUs, whose runtime-configurable TSO mode enables a direct comparison with native Arm mode. We find that Apple silicon TSO mode preserves Arm weak-memory optimizations, typically yielding execution times within 3% of Arm mode across modern applications and classic benchmarks. Although some applications experience higher TSO slowdowns, we trace these to artifacts of Apple's TSO implementation rather than inherent TSO ordering constraints. Our results challenge the perception that the Arm memory model offers a significant performance advantage over TSO in Apple silicon.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**.

Keywords: Weak memory models; x86-TSO; Arm

ACM Reference Format:
Yossi Khayet and Adam Morrison. 2026. Arm Weak Memory Consistency on Apple Silicon: What Is It Good For?. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790129

1 Introduction

The *memory consistency model* (or simply, memory model) of a shared-memory system specifies the allowed observable behaviors of concurrent programs—the order in which their memory operations appear to execute—by defining which stored values each load is allowed to return [4, 27]. Modern CPU architectures feature *weak* memory models [54, 55, 61, 66, 70], which allow program behaviors where memory operations can observably execute out of program order.[1] These models emerged to specify effects of hardware optimizations that are opaque to sequential programs but become observable in a multi-processor setting, such as memory operation buffering, overlapping, and reordering [2, 3, 59].

Unfortunately, the weaker the memory model, the harder it is to reason about the model's broader range of allowed behaviors [3, 10]. This difficulty increases the effort and cost of developing, testing, and verifying both synchronization code in concurrent software and the compilers and libraries that abstract away the hardware memory model.

The memory model weakness vs. simplicity tension is epitomized by today's dominant CPU memory models: x86 TSO [70] and the weaker Arm model [54, 61]. Program behaviors prohibited by TSO but allowed by Arm (and similar models [55, 66]) cause bugs in systems software [21, 49–51, 77], pose scalability challenges for program verification [31, 44, 62], and contribute to the complexity and open problems in programming language memory models [5, 6].

For the software community, the complexity burden imposed by memory models weaker than TSO is justifiable only if the weaker models enable performance that would be unattainable under TSO. Yet academic research on hardware TSO optimizations [13, 23, 26, 33, 67–69] suggests that this justification may not hold—that an optimized TSO implementation can perform comparably to weaker memory models [80, § 6]. It is not clear, however, that these results translate to commercial-grade CPUs. This is because the proposed hardware TSO optimizations were evaluated in simulators [13, 23, 26, 33, 67–69] or on FPGA prototypes [80], which suffer from inherent limitations of evaluation fidelity and workload scale [73]. Moreover, due to tight timing, area, and energy constraints, it is uncertain if the proposed TSO optimizations can be realized in commercial multi-GHz CPUs.

Ultimately, despite the research on efficient TSO implementations, existing weak memory architectures retain their weak models, and the clean-slate RISC-V architecture adopted an Arm-like weak model. Developers have not pushed for adoption of TSO either, presumably because they perceive weaker models as offering greater performance.

This paper critically examines that perception.

[1] "Weak" is in contrast to the strong sequential consistency (SC) model [46], which allows only executions where memory operations appear to execute sequentially in some order consistent with each thread's program order.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790129

1.1 Overview and contributions

We compare the performance obtained under TSO and the Arm memory model *directly*, on the same commercial CPU baseline: Apple silicon. Our comparison uses the runtime-configurable TSO mode of Apple's M Series CPUs, a feature originally introduced to support code generated by Apple's Rosetta x86-to-Arm binary translator [35]. Comparing the models on the same hardware avoids the limitations outlined above of prior work on hardware TSO optimizations.

We are not the first to compare the Arm and TSO models on Apple silicon; prior work [8, 76] has reported that TSO on the M1 CPU can slow down the SPEC CPU 2017 FP and Geekbench suites by an average of 8.94% and 22%, respectively. What conceptually distinguishes our work is that we treat the comparison as testing the hypothesis suggested by research on efficient TSO implementations: that TSO *need not* be inherently slower than weaker models [80, § 6]. We thus aim to *explain* the apparent contradiction between the hypothesis and observed slowdowns.

Our analysis reveals that observed TSO slowdowns stem from artifacts of Apple's TSO implementation that *are not rooted in TSO's ordering requirements*, and can therefore be addressed in future hardware. (In fact, we find TSO-specific hardware improvements across M Series CPU generations.) Even if not fixed in silicon, the hardware quirks we identify can be avoided with simple code changes. Overall, contrary to prior studies [8, 76], we conclude that *TSO on Apple silicon can be as fast as the Arm memory model.*

Central hypothesis We contend that the Arm memory model does not have an *inherent* performance advantage over TSO. Our hypothesis is that if—in the spirit of existing TSO optimization proposals [13, 23, 26, 33, 67–69]—Apple silicon TSO mode retains the CPU's Arm weak memory optimizations whenever it detects that they do not cause a *program-observable* TSO violation, then its performance should be comparable to the native Arm mode on real-world applications. This is because the CPU would disable the Arm optimizations only when it detects conflicting shared-memory accesses[2] that might observe a TSO violation [30], but we assume shared-memory conflicts are infrequent or absent in most applications, since (1) modern concurrent applications should minimize shared-memory conflicts due to the cache coherence serialization and latency penalties they incur, which impede multi-core scalability [11, 12, 20, 29, 57]; and (2) shared-memory conflicts cannot occur in sequential applications that do not utilize multiple cores.

Our hypothesis is refuted by an application slowdown in TSO mode [8, 76] only if the slowdown occurs despite all our assumptions holding. That is, if the hardware's TSO mode attempts to optimize the execution as aggressively as Arm mode, but fails because the application can actually observe TSO violations as a result of the optimization. Consequently, we must analyze any TSO slowdown to determine whether it is inherent to TSO or caused by a violation of our assumptions, in which case it might be addressable. For instance, the CPU might omit certain Arm optimizations in TSO mode or disable them too conservatively, which could be addressed in hardware. Or the application might violate our assumptions that shared-memory accesses capable of observing a TSO violation are rare, which could be fixed in software.

[2]A shared-memory *conflict* involves two concurrent memory operations to the same location, at least one of which is a write.

Contributions We conduct three studies to test our hypothesis: ① Confirming that Apple silicon TSO mode performs Arm weak memory optimizations. ② Evaluating application performance in TSO mode for a broad range of applications. ③ Understanding the root cause of observed TSO slowdowns and if they are inherent to TSO or are addressable.

① Apple's TSO implementation (§ 4) Using targeted microbenchmarks, we show that TSO mode in M Series CPUs performs load–load and store–store reordering—both prohibited by TSO—when the reordering is not program-observable. Although Intel CPUs employ a similar load–load reordering mechanism [63, § 7], our experiments reveal that Apple's TSO violation detection logic is *more precise than Intel's*, enabling more aggressive optimization.

② Performance in TSO mode (§ 5) We compare execution times in Apple silicon TSO and Arm modes, in single- and multi-core executions, across 49 applications, including popular modern software as well as classic SPEC CPU 2017 [18], PARSEC, and SPLASH-2 benchmarks [79]. Excluding seven outlier applications with severe TSO slowdowns—which we investigate below—the harmonic mean [25] of per-application average TSO slowdowns is 1.9%, with most applications exhibiting slowdowns of at most 3%. The maximal non-outlier average TSO slowdown is 11.5%.

③ Root-cause analysis of TSO slowdowns (§ 6) We manually analyze the severe TSO slowdowns to determine whether they are inherent to TSO or stem from violations of our assumptions. We find that the slowdowns arise from artifacts of Apple silicon TSO mode that are not inherent to TSO's ordering constraints. Specifically, in TSO mode: (1) effective memory-level parallelism can be limited when self-conflicts in the L1 cache trigger spurious memory-ordering squashes; (2) the CPU appears to handle partial store-to-load forwarding pairs in the instruction window serially rather than in parallel, reducing their throughput; and (3) certain vector load (SIMD) instructions exhibit slowdown on the M1 CPU. We argue that these issues are addressable in hardware; in fact, we find that the M4 already resolves the SIMD slowdown. Moreover, the slowdowns can be mitigated with simple software changes. With these software changes, the harmonic mean of per-application TSO slowdowns across our *entire* benchmark suite decreases from 4.2% to 1.9%.

1.2 Takeaways and limitations

Debates on the merit of weak vs. stronger memory models date back over 25 years [30, 33], but their claims have been limited to simulations [13, 23, 26, 30, 33, 67–69] or FPGA prototypes [80]. This paper is the first to support the arguments in favor of TSO in practice, on a commercial multi-GHz CPU. In particular, we show that TSO slowdowns should not be attributed to TSO ordering requirements without proof.

However, our study has several limitations. With respect to Apple silicon CPUs, we analyze only single-die CPUs and use execution time as the sole performance metric, so our conclusions may not extend to multi-die chips or to other metrics such as energy efficiency. More generally, M Series CPUs are aggressively superscalar, out-of-order, and speculative; memory model trade-offs could differ on simpler microarchitectures. Finally, there may be real-world application behaviors not captured by our application suite.

We hope future work explores these aspects, toward obtaining a nuanced picture of weak memory model trade-offs.

2 Background

2.1 Apple silicon

Apple silicon is a family of Arm64-based system-on-a-chip (SoC) designs, encompassing the A Series and M Series chips. A Series chips are primarily used in iPhones. M Series chips are used in desktop and laptop computers, iPad tablets, and the Vision Pro mixed-reality headset.

Apple silicon chips employ a big.LITTLE architecture [53], which combines two types of CPU cores: high-performance (P) cores and energy-efficient (E) cores. The cores are arranged into several clusters, each containing multiple cores of the same type. The number of clusters and the number of cores per cluster vary across chip models. Each core has private L1 instruction and data caches and all cores within a cluster share a common L2 cache.

Both core types utilize an aggressive superscalar, out-of-order (OoO), speculative execution microarchitecture [36]. M1 and M4 P cores can sustain issuing 8 and 10 instructions per cycle, respectively [36]. An M1 P core can maintain about 130 loads and 60 stores in flight [39], with the M4 increasing these capacities by roughly 10%–15% [16]. The CPUs are documented as employing control-flow and memory dependence speculation [36], and reverse engineering studies have additionally shown that they perform load output prediction [40], load address prediction [41], and data memory-dependent prefetching [15, 72].

Memory model configuration To support efficient execution of binaries created by Apple's Rosetta x86-to-Arm binary translator [35], an M Series CPU core can be configured to use x86's TSO memory model (§ 2.2). The core's memory model is determined by bit #1 (`EnTSO`) of its auxiliary control register (`ACTLR_EL1`); the model is TSO if and only if this bit is set. The memory model can be configured only by code running in a privileged execution level, such as the kernel.

2.2 Memory models: TSO vs. Arm

A hardware memory model specifies the allowed multi-core executions out of all theoretically possible executions, where an *execution* is modeled as a set of memory operations and a partial order $\prec_{po}$ called the *program order*, which, for each core, forms a total order specifying the core's logical execution sequence (i.e., the committed instructions).

Memory models are typically defined operationally or axiomatically. An *operational* definition describes the model in terms of an abstract machine. An *axiomatic* definition characterizes the set of valid executions using logical constraints.

Below, we summarize the TSO and Arm models. For simplicity, we focus on read/write semantics and do not discuss atomic read-modify-write (RMW) and barrier operations.

Total store ordering (TSO) [37, 70] TSO models the effect of processor-private FIFO store buffers. The store buffer holds each store's data after the store instruction commits, until the store is *performed*—written to the memory subsystem—which enables hiding the latency of the write. Store buffering weakens the order between stores and subsequent loads to different addresses: a load may be performed (read from memory) before prior stores have been performed.

Operationally, the abstract TSO machine consists of a set of sequential threads that interact through a memory subsystem, which contains one unbounded FIFO store buffer per thread [70]. Writes are enqueued into their thread's store buffer and propagated to the memory at a (non-deterministic) later time. Reads are satisfied from the youngest matching store in the thread's buffer, if any, or from memory otherwise.

Axiomatically, TSO accepts an execution if and only if there exists a global total *memory order* $<_m$ such that (1) for every pair of operations $(o_1, o_2) \neq (\text{write}, \text{read})$, if $o_1 \prec_{po} o_2$ then $o_1 <_m o_2$; and (2) every read r returns the value of the maximal matching write w in the memory order for which $(w <_m r) \vee (w \prec_{po} r)$ [59].

A scenario where $o_1 \prec_{po} o_2$ but $o_2 <_m o_1$ is called an $\mathbf{o_1}$–$\mathbf{o_2}$ ***reordering***: the two memory operations are performed in memory out of their program order. TSO thus admits store–load reordering.

Arm memory model [61] The Arm model is more complex than TSO; we only sketch its definitions, providing the details relevant to this paper. We focus on the revised Arm v8 model, which can be defined axiomatically as accepting executions with a global memory order $<_m$ and permitting all types of reorderings between data-independent operations.[3] (Operationally, it is defined by an abstract machine that explicitly models speculative and OoO execution [61].)

[3] Earlier versions were *non-multicopy-atomic*: no memory order was required and writes could become visible to different threads in different orders.

While the Arm model formally permits load–store reordering, such reordering is not implemented in practice and, in fact, is explicitly prohibited in Cortex processors as of Cortex-A76 [47, § 1]. Presumably, this is because a load being able to read from memory after a younger store (later in program order) is performed implies either out-of-order instruction commit or stores updating memory speculatively, both of which are complex mechanisms that are not featured in commercial CPUs.

3 Motivation

Our goal is to evaluate the extent to which the Arm memory model *inherently* improves performance compared to TSO. This is a question that, until now, could not be studied on commercial CPUs, which have not supported multiple memory models. The debate on the merits of weak versus stronger memory models dates back over 25 years [30, 33], but it remains relevant today because Arm-like hardware memory models can be viewed as exerting a "complexity tax" on software development, testing, and verification.

In development, performance-critical code often requires using memory-ordering primitives (rather than higher-level constructs such as locks). To illustrate, Figure 1 shows the absolute and relative numbers of Linux kernel lines of code (LOC) issuing memory-ordering primitives in each year of the last decade. New code with memory-ordering primitives is continually added as the kernel grows, and its proportion remains roughly constant, akin to an implicit "memory-ordering tax." The problem is that manual placement of memory-ordering primitives requires reasoning about possible program behaviors, which makes it a complex and error-prone task on Arm-style weak models. Indeed, program behaviors prohibited by TSO but allowed by Arm (and similar models, such as RISC-V [66] and POWER [55]) have caused bugs in systems software [21, 49–51, 77]. Similarly, model checkers face scalability challenges under Arm-style weak models due to the many possible allowed behaviors [31, 44, 62].

Programming language (PL) memory models [7, 19, 56] are designed to abstract away the hardware memory model and could, in principle, address these issues. In practice, however, their performance-critical fragments (e.g., relaxed atomics in C/C++) model behaviors allowed on commercial CPUs. As a result, they expose Arm-style reorderings, effectively reintroducing the original problem. In fact, it has been shown that ruling out certain Arm behaviors can simplify PL memory models, thereby simplifying model checking [42, 43, 45, 47].

For the software community, the complexity tax of Arm-style weak models is justifiable only if they enable performance that would be unattainable under TSO. It is therefore important to test whether this justification holds in practice.

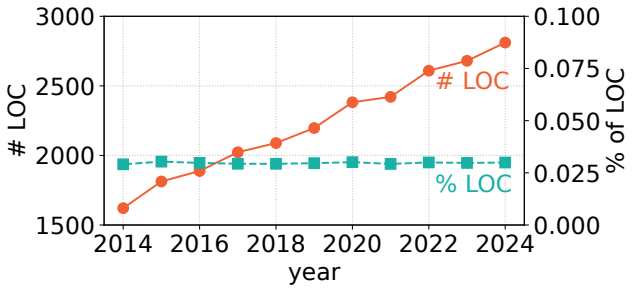


Figure 1. Linux kernel LOC calling memory-ordering primitives over time. The proportion of code using memory ordering remains roughly constant as the kernel grows over time, akin to an implicit "memory-ordering tax."

4 TSO Optimizations in Apple Silicon

This section shows that Apple silicon TSO mode performs load–load reordering (§ 4.1) and store–store reordering (§ 4.2), which are permitted by the Arm model but forbidden by TSO, when they do not result in an observable TSO violation. We establish these properties using targeted microbenchmarks, designed to probe the CPU's behavior in both memory model modes. We do not test for load–store reordering, which is also allowed by the Arm model, because it is incompatible with in-order instruction commit (see § 2.2).

We leave evaluation of TSO mode's impact on the performance of atomic RMW and memory barrier operations to future work, as we assume that modern concurrent software will not be bottlenecked on synchronization. Our subsequent application benchmarks (§ 5) support this assumption.

Experimental setup We use the M4 CPU described in § 5.1 running macOS Sequoia (version 15.5). We use Apple's Kernel Debug Kit (KDK) to boot macOS with a development kernel, which supports pinning programs to specific CPU cores. We use the PACMAN kernel patcher [65] to modify the development kernel binary so that it enables user-space access to the core cycle count register (`S3_2_c15_c0_0`). To toggle the CPU between Arm and TSO modes, we use the TSOEnabler kernel extension [38]. To rule out artifacts from hardware data memory-dependent prefetching and load address/value speculation, we verify that results remain unchanged when disabling these mechanisms via the CPU's data-independent timing (DIT) mode [54, § C5.2.4].

4.1 Load–load reordering

Load–load reordering is important for performance, as it enables an OoO core to continue performing loads even when older loads (earlier in program order) have yet to complete, e.g., due to long-latency cache misses or address computations. Consequently, a fundamental TSO implementation optimization is to execute loads out of program order speculatively, with the CPU squashing the execution upon detecting a potential TSO violation [59, 80]. Efficiency improves with

the precision of TSO violation detection, as false positives result in unnecessary pipeline flushes.

The only load–load reordering that can lead to an observable TSO violation is one in which two loads executed out of order by the core, $r(A) \prec_{po} r(B)$, interact with two stores from other processors, $w(B)$ and $w(A)$, such that their global memory order ends up being: $r(B) <_m w(B) <_m w(A) <_m r(A)$ [67]. In other words: $r(B)$ executes first (out of order) and reads from B a value that is later overwritten by $w(B)$, $w(A)$ is memory-ordered after $w(B)$ (e.g., because $w(B) \prec_{po} w(A)$), and $r(A)$ reads the value stored by $w(A)$.

Consequently, Intel CPUs are known to implement a conservative TSO violation detection mechanism that squashes a load (and all younger instructions) if the cache line it reads gets invalidated from the core's L1 cache before the load commits [63, § 7], e.g., as occurs when $w(B)$ invalidates B's cache line in the above scenario. But this is an over-approximation, as not every such invalidation implies a TSO violation.

Analysis of Apple's TSO mode We use a series of four microbenchmarks to progressively reverse engineer how Apple silicon TSO mode performs loads. Each benchmark consists of a specific pattern of shared-memory loads, designed so that whether its execution time in TSO and Arm modes differs reveals an aspect of the CPU's TSO implementation. We execute each pattern 1 M times, with each execution bracketed by serializing barrier instructions to guarantee that the instruction window contains only the tested pattern.

We now describe the patterns and analyze the test outcomes, which are presented in Figure 2. The figure shows the normalized pattern execution time histograms on M4's P cores,[4] as well as the fraction of executions that incur a TSO violation squash on an Intel Sapphire Rapids CPU, as measured by the `MACHINE_CLEARS.MEMORY_ORDERING` performance counter. In the figure and below, instruction subscripts denote both program order (i.e., $r_1 \prec_{po} r_2$) and target cache line (i.e., r_i and w_i have the same target), $\rightarrow$ denotes execution order, and $||$ denotes execution on different cores.

- $r_2 \rightarrow r_1$: Loads r_1 and r_2 are designed to execute out of order, by making r_2's operand immediately available whereas r_1's is slow to compute. The pattern's execution time is identical in Arm and TSO modes, implying that TSO mode executes the loads out of order as in Arm mode.
- $r_2 \rightarrow r_1 \,||\, w_1 \rightarrow w_2$: We add conflicting writes on another core that execute in opposite order of the loads. In Arm mode, this results in observable TSO violations (determined by inspecting the values read by the loads) in 99.93% of the executions. (Compared to the previous test, execution time is slower because the loads now miss in the cache.) TSO mode execution time is slower than in Arm mode, which is consistent with incurring squash and re-execute events. Intel executions likewise incur TSO violation squashes.

[4] E core results are qualitatively the same.

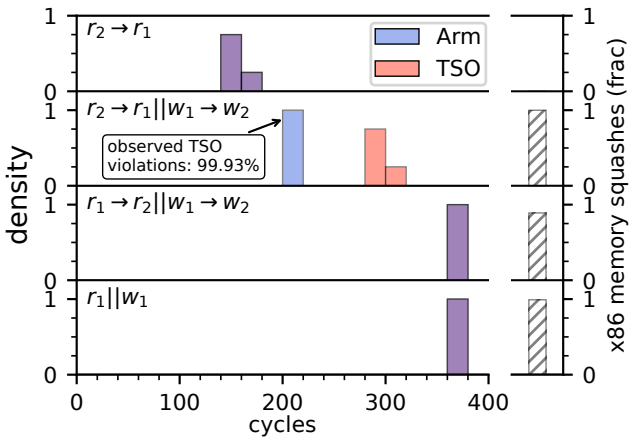


Figure 2. Apple M4 TSO mode speculatively reorders loads and detects potential TSO violations more precisely than Intel CPUs. On the M4, only the shared-memory access pattern capable of observing a TSO violation exhibits different execution times between TSO and Arm modes (left). On Intel Sapphire Rapids, additional access patterns trigger TSO violation squashes (right).

- $r_1 \rightarrow r_2 \,||\, w_1 \rightarrow w_2$: We now ensure that the load execution order matches program order by making r_2 data-depend on r_1's output, which makes TSO violations impossible [67]. As a result, the pattern's execution time is again identical in both modes. (Compared to the previous test, execution time is slower because the loads are serialized.) In contrast, the pattern incurs TSO violation squashes on the Intel CPU due to its over-approximating detection.
- $r_1 \,||\, w_1$: Here we consider a single load r_1 whose commit gets delayed after execution by preceding non-memory instructions, which enables a conflicting write to invalidate the cache line before r_1 commits. The pattern executes with identical timing in the M4 TSO and Arm modes, but incurs TSO violation squashes on the Intel CPU. This proves that Intel's TSO violation detection mechanism does not consider whether load reordering is even possible, but only whether a load's cache line remains valid upon commit.

Load–load findings: Apple silicon TSO mode speculatively reorders loads, squashing a load that had executed out of order if its data gets invalidated from the L1 cache. This detection is more precise than Intel's, enabling a more aggressive optimization.

4.2 Store–store reordering

Because the Arm memory model permits store–store reordering, Arm CPUs can optimize store performance in two key ways. First, they can coalesce non-consecutive stores to the same target into a single store buffer entry. This reduces both store buffer capacity pressure and write traffic to the L1 cache. Second, they can propagate stores to the L1 cache out of order. This further reduces store buffer capacity pressure

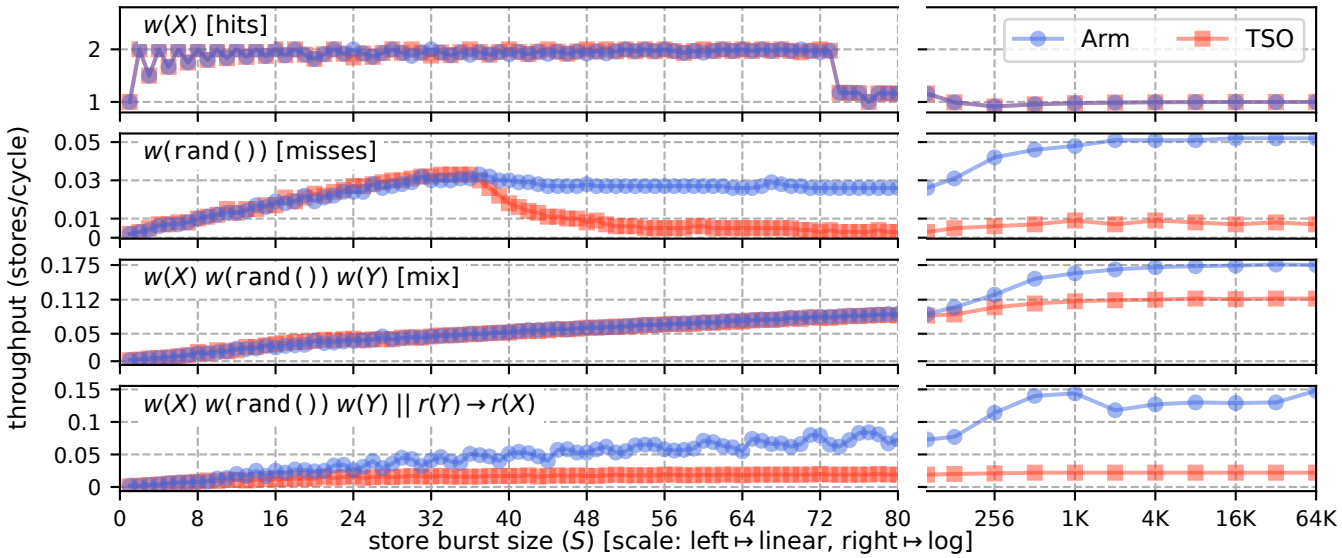


Figure 3. M4's P core TSO mode can reorder L1-hitting stores. L1-hitting stores are equally fast on TSO and Arm modes (*hits*). Bursts of > 36 L1-missing stores are slower in TSO mode (*misses*), but the gap disappears or shrinks when bursts contain both hits and misses (*mix*), implying that *hits are performed ahead of older misses*. The effect disappears when a remote core attempts to observe these reorderings (bottom), indicating TSO violation detection that incurs a performance penalty.

because stores that hit in the L1 cache can be performed without waiting if older, cache-missing stores are pending.

In contrast, conventional TSO processors employ FIFO store buffers and can coalesce only consecutive stores [59], as TSO forbids store–store reordering. Store bursts or cache-missing stores can therefore lead to a full store buffer, which blocks the dispatch of new stores and results in instruction execution stalls [13]. This problem has motivated research on preserving TSO with non-FIFO, coalescing store buffers by exposing groups of stores to the memory subsystem atomically, making intra-group reorderings unobservable. These proposals achieve atomicity either speculatively, using transactional memory-style mechanisms [9, 14, 30], or non-speculatively, by temporarily blocking remote cache coherence requests [13, 68, 74].

Analysis of P core TSO mode We test whether the M4 P core's TSO mode similarly optimizes store execution by comparing the throughput (stores per cycle) of store bursts with varying lengths and compositions in TSO and Arm modes. Each experiment consists of 100 M stores divided into bursts of size S, with each burst bracketed by serializing barrier instructions. We vary the store pattern composing the bursts and, for each pattern, the values of S. We describe the patterns and analyze the test outcomes as we walk through Figure 3, which shows the median store throughput for each burst size on the P cores.

- $w(X)$: Each burst consists of S stores to the same address X, which thus hit in the L1 cache. Both modes commit about two stores per cycle up to $S = 72$, with throughput subsequently dropping to one store per cycle. The drop is consistent with S exceeding the P cores' reported 72-entry store queue [16]. This experiment shows that TSO mode does not automatically reduce the CPU's store throughput.
- $w(\texttt{rand}())$: Each burst consists of stores to random cache lines within a 64 MB buffer, which exceeds the P core cluster's 16 MB shared L2 cache [16]. To prevent hardware prefetching, the bursts cycle through a random permutation of the buffer's cache lines,[5] so each store is an L2 miss. Both modes maintain equal throughput up to $S = 36$. Beyond this point, their throughput diverges: TSO mode throughput converges to 0.007 stores per cycle and Arm mode to 0.05 stores per cycle (an $\approx 7\times$ difference). This behavior is consistent with the P core being able to sustain 36 concurrent store misses, with a miss handling register (MSHR) released once the initiating store is performed and its store buffer entry is freed. As a result, in TSO mode, MSHRs are freed in FIFO order, which causes head-of-queue blocking and reduces throughput compared to Arm mode, which can release and reuse any MSHR as soon as its cache line arrives.
- $w(X)\ \ w(\texttt{rand}())\ \ w(Y)$: Here, we sandwich each L2-missing store between two L1-hitting stores to different cache lines. Assuming a conventional store buffer, each mode's throughput should improve by at most 3× compared to the previous experiment: if the L1-hitting stores are performed instantaneously, a burst of S stores gets performed in the time it takes to perform the $S/3$ L2-missing stores. However,

[5] This is achieved by generating target cache lines with a linear congruential generator whose period equals the number of cache lines in the buffer.

while Arm mode throughput indeed improves by up to 2.93×, TSO mode achieves a 12× improvement. It thus matches Arm mode's throughput up to $S = 80$ and falls short by at most 33% beyond that. TSO mode's behavior implies *it is employing a non-conventional TSO optimization in these executions*. We hypothesize that the CPU performs L1-hitting stores out of order, ahead of older L2-missing stores, which increases the density of L2-missing stores in the store buffer and improves their concurrency and thus overall throughput. If this hypothesis is correct, the CPU must ensure such reorderings are unobservable; we test this aspect next.

• $w(X)\ w(\texttt{rand}())\ w(Y)\,||\,r(Y) \rightarrow r(X)$: We introduce reads on another core that conflict with the constant-target stores, turning some of them into L1 misses. Due to these additional misses, Arm mode experiences a harmonic mean throughput degradation of 16%. In contrast, TSO mode's throughput degrades by a more severe harmonic mean of 3× and, for large values of S, by as much as 5×. This is consistent with the CPU detecting a potential TSO violation and disabling the optimization. In further support of this explanation, we find (results not shown) that (1) if the remote core reads only one of the store targets ($||\,r(X)$ or $||\,r(Y)$) then *there is no impact on TSO mode store throughput*, presumably because such a load cannot observe a TSO violation; and (2) reducing the frequency of the two remote reads reduces the impact on TSO mode store throughput, which is consistent with the CPU re-enabling the optimization.

These observations are consistent with both speculation- and non-speculation-based optimization mechanisms. Because there is no public information on Apple silicon utilizing post-commit speculation, we suspect that it uses a non-speculative mechanism for store reordering. We leave full reverse engineering of the mechanism to future work. Such work might also find why the optimization appears to reorder only L1-hitting stores, whereas research designs can non-speculatively reorder L1-missing stores as well [13, 68].

Analysis of E core TSO mode On the E cores, our experiments do not find evidence for store–store reordering. As in the P core, both TSO and Arm modes maintain the same throughput when all stores hit in the L1 cache. However, for all patterns, Arm mode throughput is consistently 2× better than TSO mode's. TSO mode also has no precipitous throughput drop when introducing remote reads to the $w(X)\ w(\texttt{rand}())\ w(Y)$ experiment.

Store–store findings: P core TSO mode can perform L1-hitting stores ahead of older L1-missing stores, while preventing this TSO violation from being program-observable. E core TSO mode does not employ this mechanism.

5 Benchmarking Apple Silicon TSO Mode

Having shown that Apple silicon TSO mode employs Arm-style reordering optimizations, yet still underperforms Arm mode on certain microbenchmarks (§ 4), we now examine how TSO mode impacts real-world application benchmarks.

	M1	M4
Cores	8 (4 P + 4 E)	10 (4 P + 6 E)
P core	L1D$: 128 KiB, 8-way	L1D$: 128 KiB, 8-way
	L2$ (shared): 12 MiB, 12-way	L2$ (shared): 16 MiB, 16-way
E core	L1D$: 64 KiB, 8-way	L1D$: 64 KiB, 8-way
	L2$ (shared): 4 MiB, 16-way	L2$ (shared): 4 MiB, 16-way
DRAM	LPDDR4X-4266	LPDDR5X

Table 1. Evaluated Apple silicon CPUs: M1 and M4.

5.1 Experimental setup

Hardware We conduct our experiments on two Apple M Series CPU generations, the M1 (launched in 2020) and the M4 (launched in 2024). Table 1 summarizes their specifications.

Benchmarks We use a suite of 49 parallel and sequential applications to comprehensively assess the real-world performance impact of the memory model. We start with (1) the 23 sequential applications from the industry-standardized SPEC CPU 2017 suite [18], of which 10 also have OpenMP-based parallel versions; (2) nine applications from the PARSEC 3.0 suite of multi-threaded applications spanning diverse domains [79]; and (3) eight applications from the SPLASH-2x suite, which augments the SPLASH-2 programs [75] with inputs of modern scale.[6] To complement these classic suites, we add nine modern parallel programs—including popular applications such as `ffmpeg` and `zstd`—from the `Multi-Core` and `CPU massive` test suites of OpenBenchmarking [58], a collaborative open benchmarking platform. Table 2 describes our complete benchmark suite.

We use the largest available input for each benchmark[7] except for SPEC CPU 2017 on E cores, where we use the medium-size `train` input to obtain reasonable run times (similar to P-core largest-input times).

System software We evaluate the SPEC CPU 2017 programs on macOS. We use the TSOEnabler kernel extension [38] to configure the memory model and the `pthreads` quality-of-service (QoS) interface to control application core-type placement.

We evaluate the other benchmark suites on Linux due to compilation issues on macOS. On the M1, we natively boot Asahi Linux [1], a port of Linux to the M1 and M2 SoCs, with kernel version 6.3.0. On the M4, we boot macOS and use its Virtualization Framework to run Ubuntu 24.04.2 LTS

[6] The remaining PARSEC and SPLASH-2x applications fail to compile or to run on Apple silicon.

[7] `reference` inputs for SPEC CPU 2017, `native` inputs for PARSEC and SPLASH-2x, and benchmark-specific inputs for OpenBenchmarking.

Program	Description
SPEC CPU 2017	perlbench†, gcc†, mcf†, omnetpp†, xalancbmk†, x264†, deepsjeng†, leela†, exchange2†, namd†, parest†, povray†, blender†, bwaves, cactuBSSN, lbm, wrf, cam4, imagick, nab, fotonik3d, roms, xz
PARSEC	blackscholes, bodytrack, canneal, dedup, ferret, fluidanimate, freqmine, streamcluster, swaptions
SPLASH-2x	barnes, lu_cb, lu_ncb, ocean_cp, radiosity, radix, water_nsquared, water_spatial
	OpenBenchmarking
7zip	Compresses and decompresses a sequence of $\approx D$ MiB buffers using a D MiB LZMA dictionary, for $D = 4, 8, 16, 32$
c-ray	c-ray-fast ray tracing rendering a scene of 939 spheres arranged in a fractal pattern at 3840×2160 (4K)
dav1d	AV1 video decoding of the Summer Nature 4K video (391 MiB)
ffmpeg_{x264, x265}	vbench [52] Platform scenario, transcoding a diverse set of 10 SD-to-4K videos with sizes ranging from 100 KiB to 30 MiB, using libx264/libx265
lammps	Large-scale atomic/molecular massively parallel simulator; 20 K atom input from the HECBioSim suite [32]
stockfish	Chess engine tree-search speed benchmark, up to depth 26, using a 1024 MiB transposition hash table
tensorflow	TensorFlow-CPU CNN inference benchmark, GoogLeNet [71] architecture on 100 synthetic images, batch size=1
zstd	Zstd level-19 compression of the 200 MiB Silesia corpus [22]

† Sequential programs that do not support multi-core execution.

Table 2. Evaluated application suite and its workloads.

(kernel version 6.8.0-60-generic) in a virtual machine (VM). VM virtual CPUs (vCPUs) are implemented as high-QoS threads by the host OS, so they are scheduled to P cores by default [60]. We exploit this property to ensure that programs in the VM run on P cores by using a four-vCPU VM, so the host runs each vCPU on a distinct P core.[8] Because we cannot reliably execute vCPUs on E cores, E core evaluation under Linux is limited to the M1. We use the Linux TSOEnabler [17] kernel module to configure the memory model of all CPUs (on M1) or vCPUs (on M4), as memory model configuration from within a VM is officially supported by Apple [35].

Benchmarks are compiled with GCC 12.1.0 on the M1 and GCC 13.3.0 on the M4, using each suite's default compilation level (-O3 for OpenBenchmarking and -O2 for the rest).

5.2 Methodology

We measure the total run time for SPEC CPU 2017 and OpenBenchmarking applications and the run time of the region of interest (ROI) for PARSEC and SPLASH-2x applications, as defined by these suites. We compare these run times under both TSO and Arm memory models across four configurations: single-core and multi-core runs, for binaries compiled with and without compiler optimization (-O0).

For soundness of the statistical analysis, we make no assumptions about the benchmarks' run time distributions (e.g., assuming normality). We thus use non-parametric statistical tests to determine the number of runs required to obtain high statistical confidence and to conduct hypothesis testing. Because these statistical tests assume independent measurements, we randomize the order in which benchmarks are run. This avoids possible "ordering traps" that can occur when executing measurements in a fixed sequence, where one benchmark's execution affects the system state in ways that alter the results of later measurements [24].

To obtain reliable results, we follow recommended benchmarking practices [34] and run each benchmark a minimum of 8 times and continuing until the 95% confidence interval of its run times is within 1% of the median run time.

To determine if there is a statistically significant difference between a benchmark's run time distributions in TSO and Arm modes, we apply the Kruskal-Wallis H test, which tests the null hypothesis that the run times in both modes come from the same distribution.

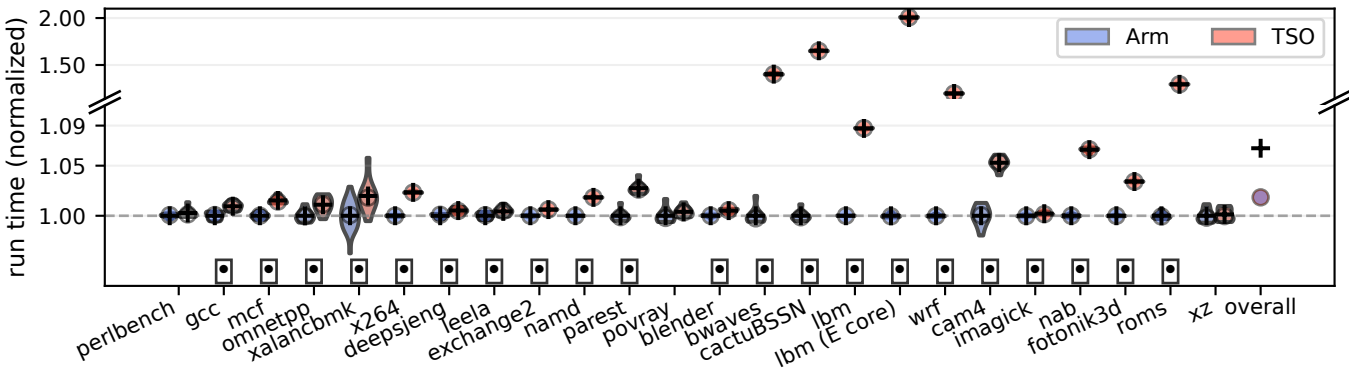
5.3 Results

We focus on the M4 P core results for applications compiled with compiler optimizations enabled. We discuss results of other configurations only if they qualitatively differ from this configuration, which occurs in only three cases.

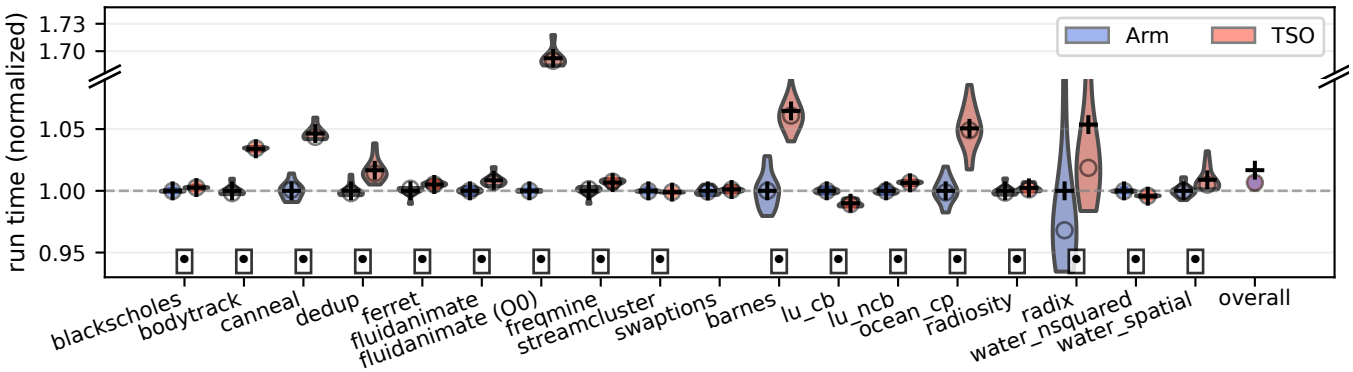
Figure 4 depicts the single-core run time distributions of our application suite. Figure 5 likewise reports four-core (the maximum possible) results. The figures account for differing run time scales across the applications by normalizing each application's run times in both modes to its average run time in Arm mode. When an application's TSO and Arm run time distributions have a statistically significant difference (determined by the H test rejecting the null hypothesis with $p < 0.05$), it is marked by a •-centered box.

While most applications exhibit statistically different run time distributions, the differences are modest regardless of whether the execution is single-core or multi-core. The average TSO slowdown is ≤ 1% for 22 of the 49 applications in single-core executions and 16 of the 36 parallel applications in multi-core executions. These numbers increase to 38/49 and 29/36 for slowdowns of at most 5%. Overall, excluding seven outliers that are discussed next, the harmonic mean of

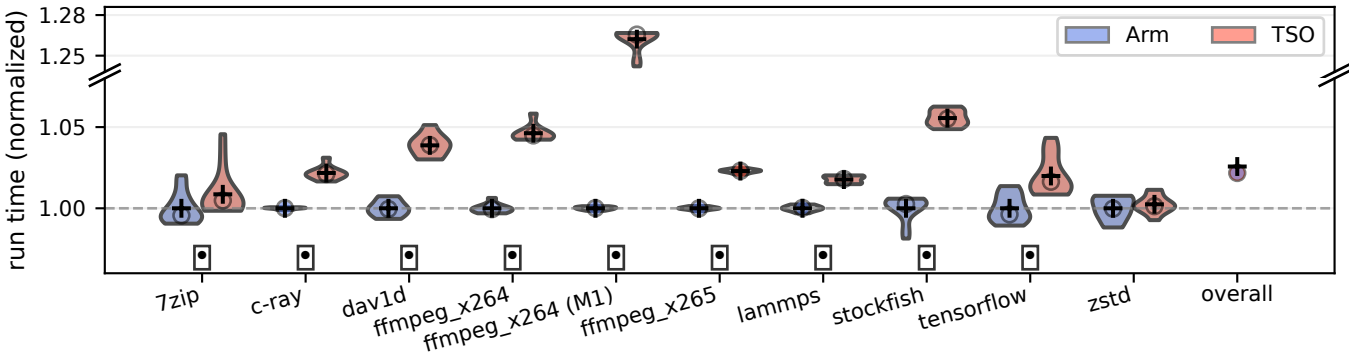
[8] We verify this using the Activity Monitor CPU monitoring tool.



(a) SPEC CPU 2017 execution on a single M4 P core.



(b) PARSEC+SPLASH-2x execution on a single M4 P core.



(c) OpenBenchmarking suite execution on a single M4 P core, and `ffmpeg_x264` on an M1 P core.

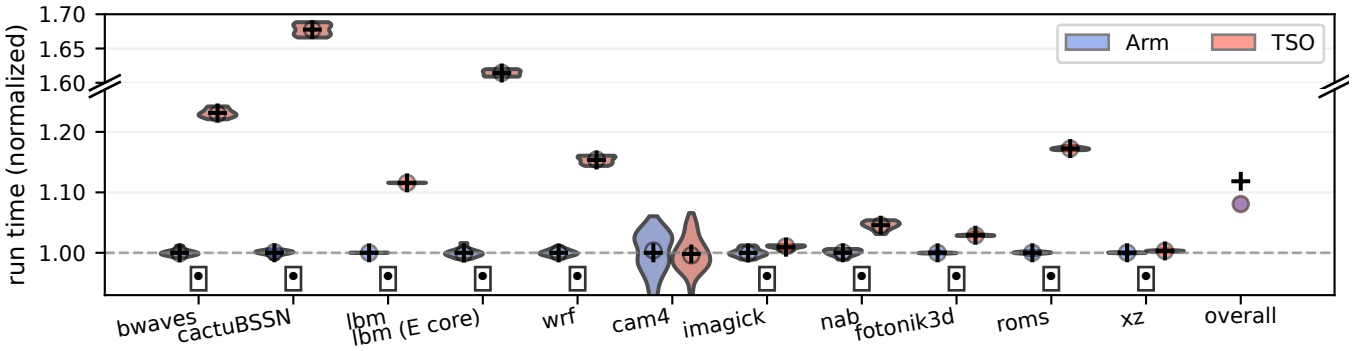
Figure 4. Application single-core run time distributions. Times are normalized to the program's average Arm-mode time. Circles and crosses denote medians and averages, respectively. Widths in the surrounding colored area indicate density of different values. •-centered boxes on the X axis denote programs whose Arm and TSO distributions have a statistically significant difference. The rightmost circle and cross show the median and harmonic mean of the per-program average normalized TSO run times, respectively.

the application TSO slowdowns is 4.1% in single-core runs and 4.3% in multi-core runs. The maximal TSO slowdowns reach 8.7% and 11.5%, respectively.

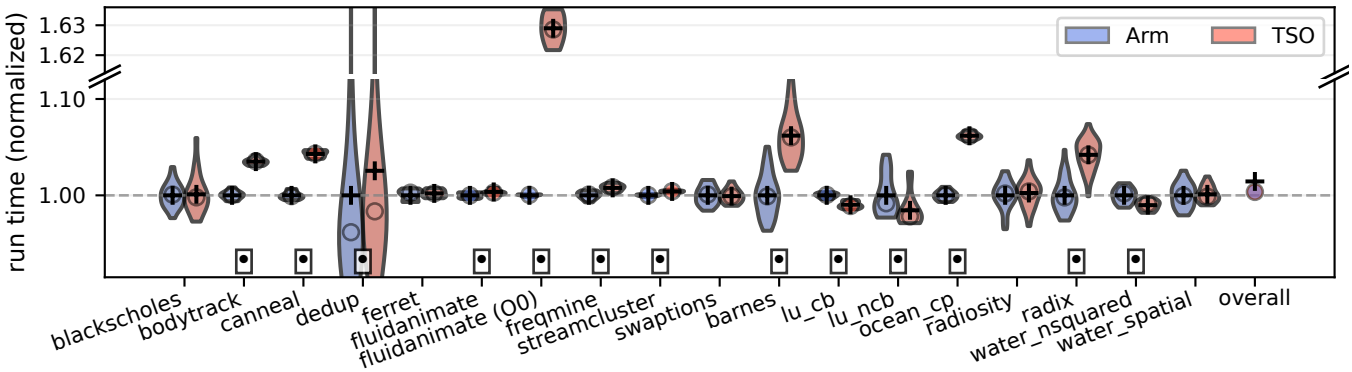
In contrast, seven outlier applications exhibit severe TSO slowdowns. On the M4, the SPEC CPU 2017 programs `bwaves`, `cactuBSSN`, `wrf`, and `roms` have average slowdowns ranging from 19% to 70% in both single- and multi-core executions, and `lbm` on E cores slows down by 2× and 1.6× in single- and multi-core executions, respectively.[9] In addition, PARSEC's `fluidanimate-O0` slows down by 70% and 63% in single- and multi-core executions, respectively. Finally, on M1 P cores, `ffmpeg_x264` slows down by 25% in single-core execution and by about 10% in multi-core execution.

However, our analysis in § 6 shows that these slowdowns are not rooted in TSO's ordering requirements—Arm mode is not faster because it allows the programs to observably

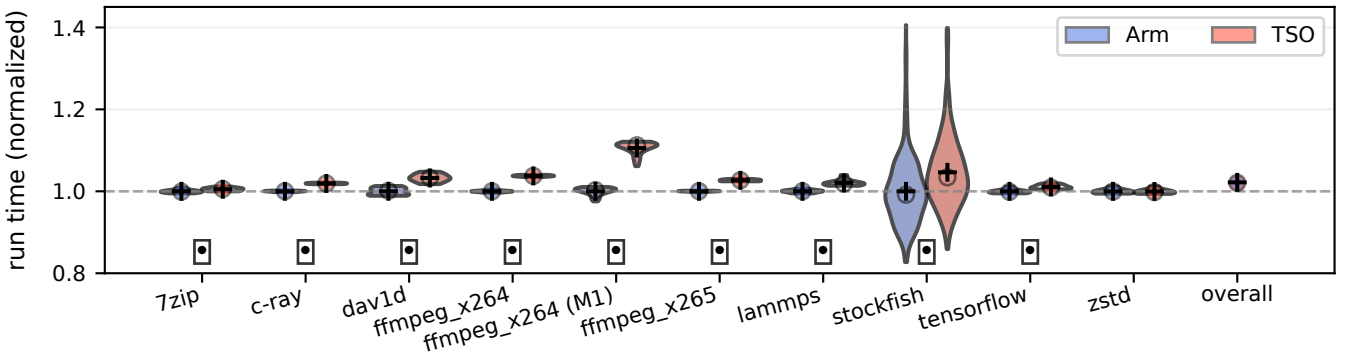
[9] SPEC CPU M4 E core and P core results are otherwise qualitatively similar.



(a) SPEC CPU 2017 execution on four M4 P cores.



(b) PARSEC+SPLASH-2x execution on four M4 P cores.



(c) OpenBenchmarking suite execution on four M4 P cores, and `ffmpeg_x264` on four M1 P cores.

Figure 5. Application multi-core run time distributions. Times are normalized to the program's average Arm-mode time. Circles and crosses denote medians and averages, respectively. Widths in the surrounding colored area indicate density of different values. •-centered boxes on the X axis denote programs whose Arm and TSO distributions have a statistically significant difference. The rightmost circle and cross show the median and harmonic mean of the per-program average normalized TSO run times, respectively.

violate TSO. (Indeed, many of the slowdowns occur in single-core executions, which cannot violate TSO.) Instead, the slowdowns stem from TSO mode implementation artifacts that can be addressed in hardware or with simple code changes. With these code changes, the harmonic mean of application TSO slowdowns across the *entire* benchmark suite and single/multi-core run types decreases from 4.2% to 1.9%.

Overall, our results suggest that in practice, program behaviors in which Apple silicon TSO mode underperforms Arm mode (§ 4) account for only a small fraction of application execution time. With respect to potential TSO violations due to load–load reordering, this is consistent with our hypothesis that conflicting shared-memory accesses capable of observing a TSO violation are rare in multi-core executions and absent in single-core executions. However, certain store burst patterns are slower in TSO mode even if no potential TSO violations occur. Our results thus appear to indicate

that such burst patterns are infrequent in the evaluated applications, but a full analysis of application store behavior is out of our scope; we leave it for future work.

In addition, future in-depth root-cause analysis of the remaining > 1% TSO slowdowns is necessary to determine if they stem from the cumulative impact of the factors explored in this paper, or from other (possibly addressable) reasons, such as atomic RMW operations and memory barriers.

6 TSO Slowdowns Root-Cause Analysis

We analyze the applications whose average TSO mode slowdown (§ 5.3) exceeds 15% to identify its root cause. Our conceptual hypothesis predicts that TSO slowdowns should arise from the CPU failing to perform Arm weak memory optimizations in TSO mode. However, with one exception, we find that the slowdowns are caused by *TSO mode implementation artifacts* that are not inherently required to enforce TSO. Specifically, these artifacts consist of:

- Reduced effective memory-level parallelism when self-conflicts in the L1 cache trigger spurious memory-ordering related load squashes (§ 6.1), which affects the SPEC CPU 2017 outlier programs.
- Serialized handling of partial store-to-load forwarding pairs in the instruction window (§ 6.2), which affects `fluidanimate-O0`.
- Slowdown of certain vector load instructions on the M1, but not the M4 (§ 6.3), which affects `ffmpeg`.

The exception is `lbm` on the M4 E cores, whose lack of store–store reordering (see § 4.2) causes the slowdown (§ 6.4).

6.1 Reduced effective memory-level parallelism

Apple silicon P and E cores speculatively reorder loads in both Arm and TSO modes, with TSO mode preventing observable TSO violations by squashing loads that had executed out of order if their data subsequently gets invalidated from the L1 data cache (§ 4.1). This design limits effective memory-level parallelism (MLP) based on the accessed addresses: a set of loads whose targets cannot co-reside in the L1 data cache due to conflicts cannot all execute out-of-order and in parallel, because loads whose data is evicted by these conflicts will be squashed.

Our analysis reveals that this effect is the root cause of the severe TSO slowdowns in `bwaves`, `cactuBSSN`, `wrf`, and `roms`. Consistently with our central hypothesis, the slowdown's cause is not inherent to TSO's ordering constraints. The cause is an artifact of Apple silicon's speculative load–load reordering mechanism, which is not *itself* required by TSO. A non-speculative TSO load–load reordering mechanism [67] would not limit MLP in this way. Moreover, even without hardware changes, software can avoid MLP-limiting L1 conflicts. We eliminate the L1 conflicts in the hot loops of the affected programs—and thereby their TSO slowdowns—by modifying the memory allocator to randomize allocated buffers' alignment. In fact, this change is not a TSO-only optimization for these benchmarks. It is beneficial regardless of the memory model, as the high L1 cache miss rate caused by the L1 cache conflicts is detrimental to performance.

	Arm	TSO
# L1 load misses / # retired loads	2.36	3.53
% of L1 load misses due to squashed loads	26%	71%
# load unit μ-ops / # retired loads	11	17.96

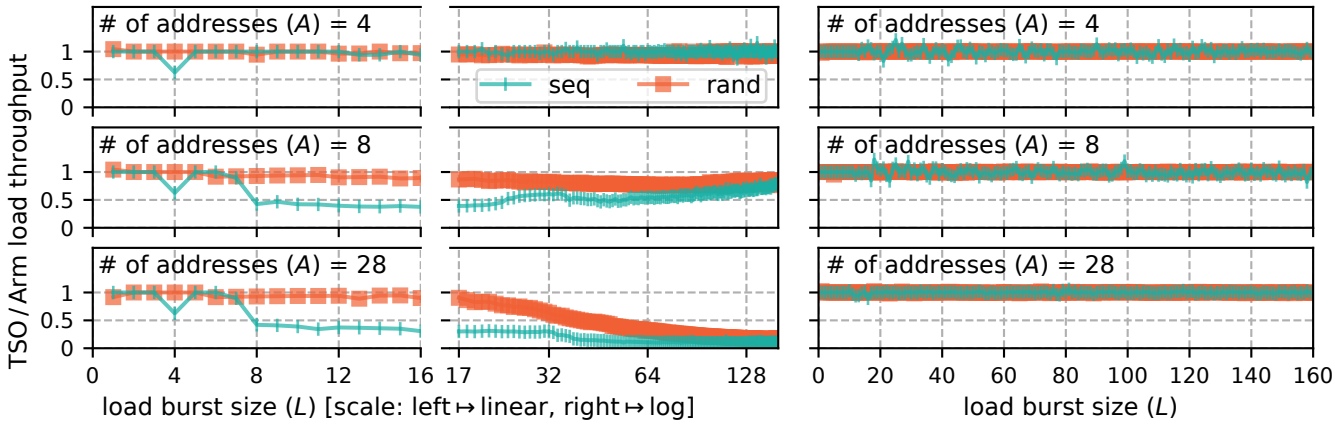
Table 3. Single-core `cactuBSSN` execution characteristics. Compared to Arm mode, TSO mode exhibits 50% more L1 misses overall, more L1 misses due to squashed loads, and more squashed loads.

Investigation We analyze `cactuBSSN`, the benchmark with the highest TSO slowdown. We find that the program exhibits the same hot loops in TSO and Arm modes, with slower execution in TSO mode. CPU performance counter measurements (Table 3) show that the slowdown is correlated with 50% more L1 load misses, 71% of which are due to loads that are eventually squashed. TSO mode also exhibits a higher rate of squashed load μ-ops (i.e., more μ-ops per retired load).

Each iteration in the benchmark's hot loops issues independent loads accessing the same offset in many (up to 50) 3D grids, which are stored in memory as 1D arrays. The addresses of these arrays all have the same intra-page offsets, so the loads in each iteration all target the same intra-page offset. Because the L1 cache in M Series CPUs is virtually-indexed [78, § 4.1], such accesses suffer from L1 cache set conflicts, suggesting that the TSO slowdown is caused by speculative out-of-order loads being squashed when their data are evicted from the L1 cache.

Analysis To test the above explanation, we use targeted microbenchmarks to analyze load throughput (loads per cycle) in loops modeled after the benchmark's hot loops. Each experiment consists of a loop traversing a set of 16 MiB buffers. Each loop iteration issues L independent loads to A distinct addresses, all at the same cache-line offset in the buffers. Buffers are statically assigned to loads in a round-robin fashion. We test two traversal patterns: sequential (`seq`), where iteration i accesses the i-th cache line in each buffer, and random (`rand`), where iteration i accesses cache line $\pi(i)$ in each buffer according to a predetermined random permutation π.

Figure 6a compares per-iteration load throughput between TSO and Arm modes on the M4 P cores (E core results are similar), for representative values of A. Consistent with our explanation, TSO mode throughput drops relative to Arm mode once the load burst accesses more than 8 addresses—exactly the associativity of the L1 data cache (see Table 1). The drop occurs in both access patterns, but is worse (and more visible in the figure) for `seq`. Comparing bursts that access all A buffers ($L \geq A$), the drops worsen as A increases.



(a) Addresses with identical intra-page offsets.

(b) Addresses with random intra-page offsets.

Figure 6. TSO mode limits MLP when self-conflicts in the L1 cache trigger frequent load squashes. Relative (TSO/Arm) load throughput of a loop traversing A buffers accessed by L independent loads in each iteration, using sequential or random access patterns. TSO throughput drops relative to Arm when the addresses have identical intra-page offsets (a) but not when the intra-page offsets are random (b).

Because the TSO slowdowns stem from underutilization of the L1 data cache, they can be alleviated through software changes that improve cache utilization. To support this claim, we repeat the microbenchmark experiments with the buffers starting at random intra-page offsets. This way, in each loop iteration, loads from different buffers target different intra-page offsets and thus target different L1 cache sets. Figure 6b shows the results: TSO mode now matches Arm mode's throughput in all configurations.

Motivated by the above results, we rerun the outlier SPEC CPU benchmark while making large (> page size) dynamically-allocated buffers start at random intra-page offsets (in cache-line granularity).[10] Figure 7 shows how this allocation policy eliminates the outlier TSO slowdowns, in both single- and multi-core executions. (Also shown is `nab`, whose single-core TSO slowdown is reduced from 6.6% to 4.1% due to this policy.) In addition, this policy also improves execution time in Arm mode—sometimes dramatically (e.g., by ≈ 40% for `cactuBSSN`). This is because the underutilization of the L1 cache that this policy alleviates is detrimental to performance regardless of the memory model.

6.2 Serialized partial store-to-load forwarding

Apple silicon P and E cores implement *store-to-load (STL) forwarding*, which enables a load to directly read an older store's value prior to the store writing to the L1 cache [36]. STL forwarding is *partial* when the load's address range is not contained in the store's, such as an 8-byte load reading from a 4-byte store. In this case, the load still needs to access the cache to obtain the rest of its data.

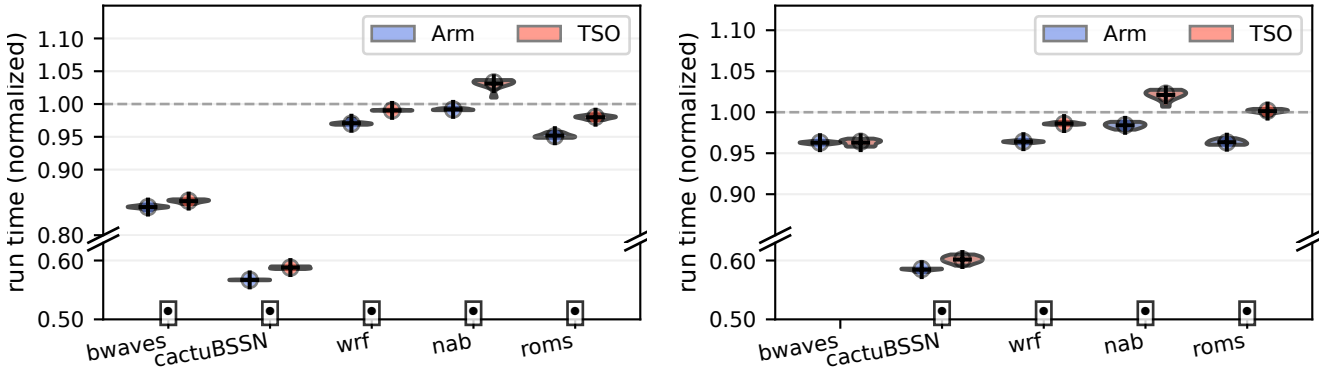
[10]This is achieved through a dynamically-interposed library that transparently pads each allocation request by one page and uses the extra space to return a buffer of the requested size with a randomized starting offset.

Our analysis of the `fluidanimate-O0` TSO slowdown reveals that partial STL forwarding is slower in TSO mode. Based on subsequent targeted microbenchmarking, we hypothesize that TSO mode handles multiple partial STL forwarding pairs in the instruction window serially rather than in parallel. However, as discussed below, we do not find a reason that TSO's ordering constraints would necessitate such an implementation.

Investigation Using the Linux `perf` tool, we identify that most stall cycles in the execution of `fluidanimate-O0` are related to the construction of a `Vec3` object, which is executed as part of a hot loop. The constructor initializes the object's three floating point fields with 4-byte stores, whereas the `-O2` version of the binary, which does not exhibit a severe TSO slowdown, uses an 8-byte and 4-byte store pair. While we confirm that changing the `-O0` constructor to use such a store pair mitigates the slowdown, we are unable to reproduce the slowdown with targeted microbenchmarking of similar store instruction patterns.

Consequently, we inspect the code flow following the constructor and find that the object is copied to a different location on the stack, with this `memcpy` operation using an 8-byte and 4-byte load pair to read the object. We identify these loads as the root cause of the slowdown by modifying the `-O0` binary, substituting this load pair with three 4-byte loads, which eliminates the slowdown (Figure 8).

Analysis We use targeted microbenchmarks to analyze the performance of different STL patterns, consisting of an r-byte load reading from a w-byte store to the same address (denoted $W{:}w\ R{:}r$), for $r, w \in \{4, 8\}$. For each pattern, we time a loop of 100 M such pairs and compare the pair's throughput (i.e., average iterations per ns) in TSO and Arm modes. We also evaluate variants in which each pair is followed by



(a) Single M4 P core. **(b)** Four M4 P cores.

Figure 7. Elimination of SPEC CPU 2017 severe TSO slowdowns and speedups due to randomized allocation intra-page offsets. M4 P core run times, normalized to each program's original average Arm-mode time (cf. Figures 4a and 5a).

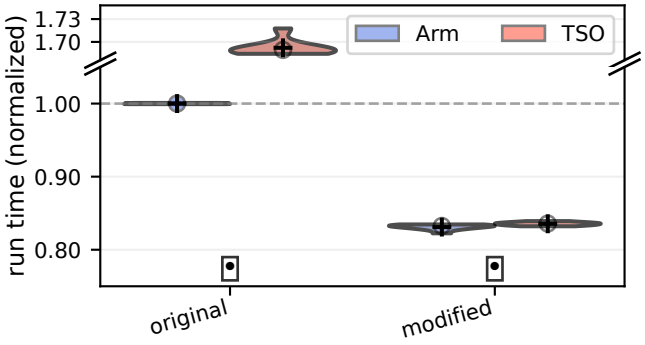


Figure 8. `fluidanimate -O0` TSO slowdown mitigated by modifying the binary to make STL pairs data sizes match. (Single M4 P core results, all normalized to the original average Arm-mode time; M1 P and E core results are similar.)

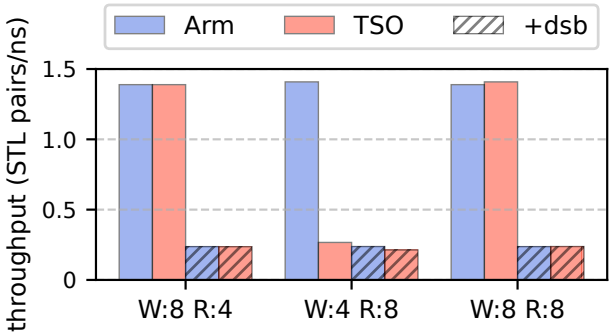


Figure 9. Throughput of STL pairs where an r-byte load reads from a w-byte store (W:w R:r) on an M4 P core. (M1 P and E core results are qualitatively identical.)

an Arm data memory barrier (`dmb`), data synchronization barrier (`dsb`), or instruction synchronization barrier (`isb`).

Figure 9 presents the results of the benchmark and its `dsb` variant, which essentially serializes the pipeline, ensuring that all prior memory operations are complete before any subsequent instruction can execute [54]. STL forwarding in TSO mode is as fast as Arm's if the load obtains all its data from the store ($r \leq w$) but is about 5× slower otherwise, when the load must also access the cache. However, when the STL pairs are serialized by a `dsb`, all patterns exhibit roughly the same throughput in both modes, which is close to the throughput of W:4 R:8 in TSO mode.

We therefore hypothesize that *TSO mode serializes partial STL pair handling*, possibly because it waits for the store to write to the L1 cache in this case (i.e., disables STL forwarding). However, *TSO does not require such a partial STL forwarding implementation*. In the common case of a load reading from a single cache line, the load can read from both the store and the cache, and verify that the cache line remains valid in the L1 cache when it commits, similarly to how the CPU already detects potential load–load reordering TSO violations (§ 4.1). We thus argue that future Apple silicon generations can address this TSO slowdown in hardware.

6.3 Slow SIMD loads on M1

Our analysis of the `ffmpeg_x264` TSO slowdown on M1 P cores reveals that it is caused by Arm NEON vector loads whose data size exceeds 16 B being slower in TSO mode, which does not happen on the M4.

Investigation Using the Linux `perf` tool, we find that several vector load instructions, which load 32 B or more, are responsible for a large fraction of the stall cycles in the execution. Following our analysis of load performance on the M1, described below, we confirm that substituting such loads with semantically-equivalent patterns that use ≤ 16 B vector loads mitigates much of the slowdown (Figure 11).

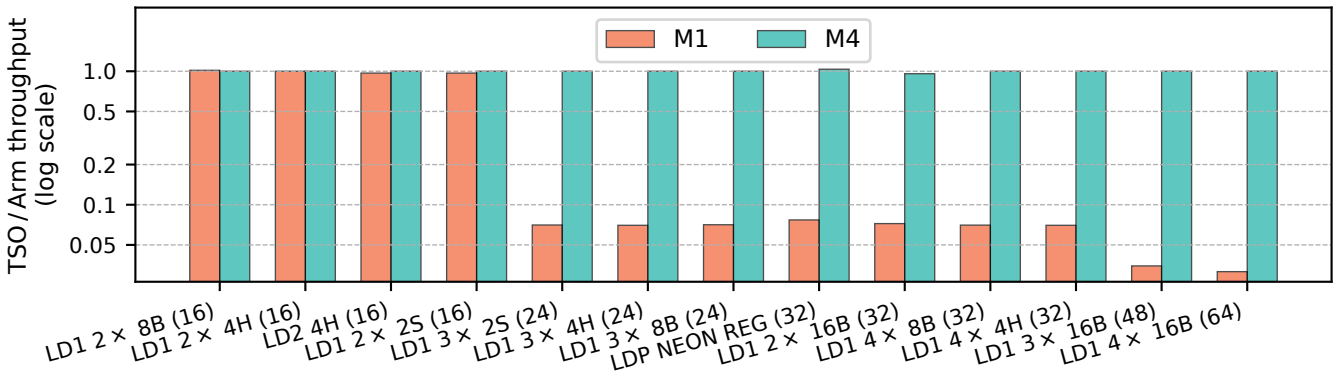


Figure 10. Relative (TSO/Arm) throughput of cache-hitting loads on M1 and M4 P cores. SIMD $v \times n\,e$ notation specifies the number v of n-element vectors loaded, where element e is B=byte, H=2 B, or S=4 B. Parenthesized numbers specify the loaded data size in bytes.

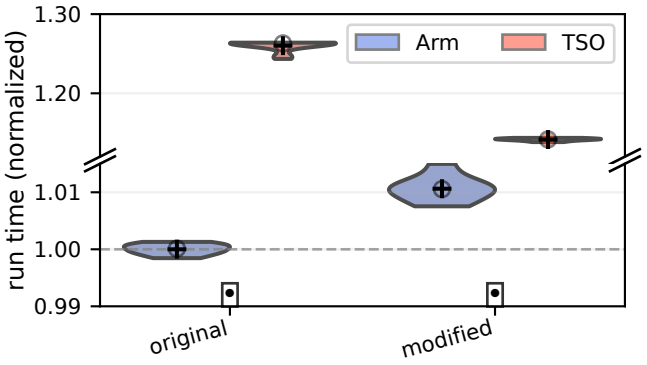


Figure 11. `ffmpeg_x264` M1 P core TSO slowdown partially mitigated by substituting 32 B vector loads with two 16 B vector loads. All times are normalized to original average Arm-mode time.

Analysis We use targeted microbenchmarks to systematically analyze the throughput of cache-hitting load instructions on Apple silicon. These include (1) single register loads (`ldr`) across all addressing modes, offset specification, and data sizes (1, 2, 4, and 8 B into general-purpose registers and 16 B into NEON registers); (2) paired loads (`ldp`) into two registers or two 128-bit NEON registers; and (3) SIMD loads, across all lane counts (how many elements are loaded), element sizes, load count (how many NEON registers are filled), and structure variants (the `ld1/2/3/4` instructions, which distribute loaded elements across multiple registers in different interleaved manners). For each instruction type, we time a loop of 100 M instances of the instruction and compare its throughput (i.e., average iterations per ns) in TSO and Arm modes, on both M1 and M4 P cores.

Figure 10 shows a representative subset of the results. On the M1 P core TSO mode, loads of $> 16\,B$ are more than 10× slower than in Arm mode, whereas loads of $\leq 16\,B$ are not slowed down. In contrast, on the M4, all loads variants have equal throughput in both modes. This shows that the M1 SIMD TSO slowdown is a hardware artifact that was addressed on the M4 and is not inherently caused by TSO constraints.

6.4 E core TSO implementation

As reported in § 4.2, store throughput in TSO mode on the E cores is half that of Arm mode. Our analysis reveals that this is the root cause for `lbm`'s TSO slowdown on E cores.

Specifically, `lbm`'s execution time is dominated by a streaming loop that performs a fused "stream and collide" operation on a 3D grid represented as a ≈ 200 MiB array. Each iteration performs a store burst, writing 19 double-precision values to cells in an output grid. This loop being 2× slower in TSO mode is consistent with E core TSO mode having half the store throughput of Arm mode. While the loop also reads from an input grid to compute written values, we experimentally confirm that it is bottlenecked by store throughput by simplifying it into a store-only loop (writing dummy values) and observing approximately the same TSO slowdown.

7 Related Work

Hill [33] and Guiady et al. [30] originally argued that strong memory models (SC or TSO) could offer comparable performance to weak models and therefore, the complexity burden of hardware weak models is not justified. Both used simulations and six benchmarks to support their claims. Likewise, subsequent work on optimized microarchitectural implementations of SC [9, 14, 26, 28, 48, 64] and TSO [13, 23, 67–69] used simulations or FPGA prototypes [80].

Wrenger et al. [76] evaluate the multi-threaded SPEC2017 CPU FP benchmark suite on the M1 TSO mode and observe an average 8.94% slowdown. Beck et al. [8] find that GeekBench in M1 TSO mode has a 22% lower score. Both works study only the M1 CPU and do not analyze the root cause of the slowdowns.

8 Conclusion

This paper is the first to challenge the perception that the Arm memory model enables higher performance than TSO based on evaluation with commercial multi-GHz CPUs. We find that Apple's TSO mode retains the CPU's Arm weak-memory optimizations, yielding execution times typically within 3% of Arm mode across a broad set of applications. Although some applications experience higher TSO slowdowns, these are not due to TSO ordering constraints but arise from artifacts of Apple's TSO implementation.

Our results suggest that TSO slowdowns should not be attributed to TSO ordering requirements without proof, and highlight the need for further research to obtain a nuanced picture of weak memory model trade-offs across additional performance metrics and microarchitectures.

Acknowledgments

We thank the paper's shepherd, Sam Ainsworth, and the anonymous reviewers for their feedback, which helped improve the paper.

This research was supported by the Israel Science Foundation under grant no. 853/24.

References

[1] 2021. Linux on Apple Silicon. https://asahilinux.org.

[2] Sarita V. Adve and Hans-J. Boehm. 2010. Memory Models: A Case For Rethinking Parallel Languages and Hardware. *CACM* 53, 8 (2010). doi:10.1145/1787234.1787255

[3] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29, 12 (1996). doi:10.1109/2.546611

[4] Sarita V. Adve and Mark D. Hill. 1993. A Unified Formalization of Four Shared-Memory Models. *IEEE TPDS* 4, 6 (1993). doi:10.1109/71.242161

[5] Mark Batty. 2014. *The C11 and C++11 Concurrency Model.* Ph. D. Dissertation. University of Cambridge.

[6] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The Problem of Programming Language Concurrency Semantics. In *ESOP* 2015. doi:10.1007/978-3-662-46669-8_12

[7] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ Concurrency. In *POPL* 2011. doi:10.1145/1926385.1926394

[8] Martin Beck, Koustubha Bhat, Lazar Stričević, Geng Chen, Diogo Behrens, Ming Fu, Viktor Vafeiadis, Haibo Chen, and Hermann Härtig. AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM. In *ASPLOS* 2023. doi:10.1145/3575693.3579849

[9] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *ISCA* 2009. doi:10.1145/1555815.1555785

[10] Hans-J. Boehm and Sarita V. Adve. 2012. You Don't Know Jack about Shared Variables or Memory Models. *CACM* 55, 2 (2012). doi:10.1145/2076450.2076465

[11] Silas Boyd-Wickizer. 2014. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[12] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. In *PPoPP* 2010. doi:10.1145/1837853.1693488

[13] Juan M. Cebrian, Magnus Jahre, and Alberto Ros. Temporarily Unauthorized Stores: Write First, Ask for Permission Later. In *MICRO* 2024. doi:10.1109/MICRO61859.2024.00065

[14] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA* 2007. doi:10.1145/1273440.1250697

[15] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *USENIX Security* 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru

[16] Jiajie Chen. 2024. Apple M4 P-core. https://jia.je/cpu/m4_pcore.html.

[17] Yangyu Chen. 2022. TSOEnabler for Linux. https://github.com/cyyself/m1tso-linux.

[18] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017 benchmark. https://www.spec.org/cpu2017/.

[19] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *PACMPL* 4, POPL (2019). doi:10.1145/3371102

[20] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS* 2015. doi:10.1145/2786763.2694359

[21] Will Deacon. 2018. locking/qspinlock: Ensure node is initialized before updating prev->next. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95bcade33a8a.

[22] Sebastian Deorowicz. 2012. Silesia compression corpus. https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.

[23] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. WeeFence: Toward Making Fences Free in TSO. In *ISCA* 2013. doi:10.1145/2508148.2485941

[24] Dmitry Duplyakin, Nikhil Ramesh, Carina Imburgia, Hamza Fathallah Al Sheikh, Semil Jain, Prikshit Tekta, Aleksander Maricq, Gary Wong, and Robert Ricci. Avoiding the Ordering Trap in Systems Performance Measurement. In *USENIX ATC 23* 2023. https://www.usenix.org/conference/atc23/presentation/duplyakin

[25] Lieven Eeckhout. 2024. R.I.P. Geomean Speedup. *IEEE Computer Architecture Letters* 23, 1 (2024). doi:10.1109/LCA.2024.3361925

[26] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *ICPP* 1991.

[27] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessor. In *ISCA* 1990. doi:10.1145/325096.325102

[28] Chris Gniady and Babak Falsafi. Speculative Sequential Consistency With Little Custom Storage. In *PACT* 2002. doi:10.1109/PACT.2002.1106016

[29] Vincent Gramoli. More Than You Ever Wanted to Know about Synchronization. In *PPoPP* 2015. doi:10.1145/2688500.2688501

[30] Chris Guiady, Babak Falsafi, and Terani N Vijaykumar. Is SC + ILP = RC?. In *ISCA* 1999. doi:10.1109/ISCA.1999.765948

[31] Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. 2024. An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic. *PACMPL* 8, POPL (2024). doi:10.1145/3632863

[32] HECBioSim. 2018. HPC Benchmarking Suite. https://www.hecbiosim.ac.uk/access-hpc/hpc-benchmarking-suite.

[33] Mark D. Hill. 1998. Multiprocessors Should Support Simple Memory-Consistency Models. *IEEE Computer* 31, 8 (1998). doi:10.1109/2.707614

[34] Torsten Hoefler and Roberto Belli. Scientific Benchmarking of Parallel Computing Systems. In *SC* 2015. doi:10.1145/2807591.2807644

[35] Apple Inc. 2025. Accelerating the performance of Rosetta. https://developer.apple.com/documentation/virtualization/accelerating-

the-performance-of-rosetta.
[36] Apple Inc. 2025. Apple Silicon CPU Optimization Guide: 4.0. https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide.
[37] SPARC International Inc. 1992. *The SPARC Architecture Manual: Version 8*.
[38] Saagar Jha. 2020. TSOEnabler. https://github.com/saagarjha/TSOEnabler.
[39] Dougall Johnson. 2021. Apple Microarchitecture Research. https://dougallj.github.io/applecpu/firestorm.html.
[40] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. FLOP: Breaking the Apple M3 CPU via False Load Output Predictions. In *USENIX Security* 2025. https://www.usenix.org/conference/usenixsecurity25/presentation/kim-jason
[41] Jason Kim, Daniel Genkin, and Yuval Yarom. SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon. In *IEEE S&P* 2025. doi:10.1109/SP61157.2025.00098
[42] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *PACMPL* 2, POPL (2017). doi:10.1145/3158105
[43] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model Checking for Weakly Consistent Libraries. In *PLDI* 2019. doi:10.1145/3314221.3314609
[44] Michalis Kokologiannakis and Viktor Vafeiadis. HMC: Model Checking for Hardware Memory Models. In *ASPLOS* 2020. doi:10.1145/3373376.3378480
[45] Michalis Kokologiannakis and Viktor Vafeiadis. GenMC: A Model Checker for Weak Memory Models. In *CAV* 2021. doi:10.1007/978-3-030-81685-8_20
[46] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (1979). doi:10.1109/TC.1979.1675439
[47] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Putting Weak Memory in Order via a Promising Intermediate Representation. *PACMPL* 7, PLDI, Article 183 (2023). doi:10.1145/3591297
[48] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS* 2012. doi:10.1145/2248487.2151006
[49] Waiman Long. 2015. locking/qspinlock: Use _acquire/_release() versions of cmpxchg() & xchg(). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=64d816cba06c.
[50] Waiman Long and Peter Zijlstra. 2015. qspinlock code at version 4.4 of Linux Kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/locking/qspinlock.c?h=v4.4.
[51] Waiman Long and Peter Zijlstra. 2020. qspinlock code at version 5.6 of Linux Kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/locking/qspinlock.c?h=v5.6.
[52] Andrea Lottarini, Alex Ramirez, Joel Coburn, Martha A. Kim, Parthasarathy Ranganathan, Daniel Stodolsky, and Mark Wachsler. vbench: Benchmarking Video Transcoding in the Cloud. In *ASPLOS* 2018. doi:10.1145/3173162.3173207
[53] Arm Ltd. 2021. Arm Big.LITTLE. https://www.arm.com/technologies/big-little.
[54] Arm Ltd. 2024. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/. ARM DDI 0487L.a.
[55] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An Axiomatic Memory Model for POWER Multiprocessors. In *CAV* 2012. doi:10.1007/978-3-642-31424-7_36
[56] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL* 2005. doi:10.1145/1040305.1040336
[57] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys* 2012. doi:10.1145/2168836.2168855
[58] Phoronix Media. 2025. OpenBenchmarking. https://openbenchmarking.org.
[59] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood, and Natalie Enright Jerger. 2020. *A Primer on Memory Consistency and Cache Coherence* (2nd ed.). Morgan & Claypool Publishers.
[60] Howard Oakley. 2024. Inside M4 chips: P cores hosting a VM. https://eclecticlight.co/2024/11/13/inside-m4-chips-p-cores-hosting-a-vm/.
[61] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *PACMPL* 2, POPL (2017). doi:10.1145/3158107
[62] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *PLDI* 2019. doi:10.1145/3314221.3314624
[63] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security* 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/ragab
[64] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *SPAA* 1997. doi:10.1145/258492.258512
[65] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *ISCA* 2022. doi:10.1145/3470496.3527429
[66] RISC-V. 2024. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA. https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view. Version 20240411.
[67] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. Non-Speculative Load-Load Reordering in TSO. In *ISCA* 2017. doi:10.1145/3079856.3080220
[68] Alberto Ros and Stefanos Kaxiras. Non-Speculative Store Coalescing in Total Store Order. In *ISCA* 2018. doi:10.1109/ISCA.2018.00028
[69] Alberto Ros and Stefanos Kaxiras. The Superfluous Load Queue. In *MICRO* 2018. doi:10.1109/MICRO.2018.00017
[70] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *CACM* 53, 7 (2010). doi:10.1145/1785414.1785443
[71] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper With Convolutions. In *CVPR* 2015. doi:10.1109/CVPR.2015.7298594
[72] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *IEEE S&P* 2022. doi:10.1109/SP46214.2022.9833570
[73] Vincent M. Weaver and Sally A. McKee. Are Cycle Accurate Simulations a Waste of Time?. In *WDDD* 2008.
[74] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *ISCA* 2007. doi:10.1145/1250662.1250696
[75] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA* 1995. doi:10.1145/223982.223990
[76] Lars Wrenger, Dominik Töllner, and Daniel Lohmann. 2024. Analyzing the memory ordering models of the Apple M1. *Journal of Systems*

Architecture 149 (2024). doi:10.1016/j.sysarc.2024.103102

[77] Pan Xinhui. 2016. locking/qspinlock: Use atomic_sub_return_release() in queue_spin_unlock(). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ca50e426f96c.

[78] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. Synchronization Storage Channels (S2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In *USENIX Security* 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jiyong

[79] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. PARSEC3.0: A Multicore Benchmark Suite with Network Stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News* 44, 5, 1–16 2017. doi:10.1145/3053277.3053279

[80] Sizhuo Zhang. 2019. *Constructing and Evaluating Weak Memory Models*. Ph. D. Dissertation. MIT.