Journal of Systems Architecture 149 (2024) 103102

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Analyzing the memory ordering models of the Apple M1

Lars Wrenger *, Dominik Töllner, Daniel Lohmann

Leibniz Universität Hannover, Appelstr. 4, Hannover, 30167, Germany

ARTICLE INFO

Keywords:
TSO
Memory ordering
Apple M1
ARM

ABSTRACT

The Apple M1 ARM processor family incorporates two memory consistency models: the conventional ARM weak memory ordering and the *Total store ordering (TSO)* model from the x86 architecture utilized by Apple's x86 emulator, Rosetta 2. The presence of both memory ordering models on the same hardware enables us to thoroughly benchmark and compare their performance characteristics and worst-case workloads.

In this paper, we assess the performance implications of TSO on the Apple M1 processor architecture. Based on the multi-threading workloads of the SPEC2017 CPU FP benchmark suite, our findings indicate that TSO is, on average, 8.94 percent slower than ARM's weaker memory ordering. Through synthetic benchmarks, we further explore the workloads that experience the most significant performance degradation due to TSO. We also take a deeper look into the specific atomic instructions provided by the ARMv8.3 specification and their synchronization overheads.

1. Introduction

On traditional uniprocessor systems, the effects of memory accesses are observable in the same order as they were specified in the instruction stream (program order). This is still the case for multitasking on a single core. Challenges arise when the memory is shared between multiple participants who access it *concurrently*, such as other cores, processors, or accelerators. Providing a consistent *global order* in which memory accesses are visible to all observers can be particularly difficult for multiscalar processors with instruction reordering and local caches that buffer accesses.

Memory consistency models (MCMs) formalize how writes to shared memory can be observed by different participants within a shareability domain. These hardware-defined guarantees provide rules that lead to predictable results of shared memory operations [1–3]. The strictness of the provided guarantees varies from model to model. Even though both x86 and ARM define MCMs that allow limited instruction reordering [4–6], x86 guarantees a globally consistent order for stores (TSO). ARM, in contrast, allows stores to different memory locations to be observed differently from the program order. While complicating the programming model, ARM's weaker memory ordering allows processors to reorder instructions more freely, potentially reducing synchronization overheads between caches. Seeing this tradeoff between higher performance and simpler programming models, we ask how extensive the performance benefits actually are.

Apple's M1 processors implement the ARMv8.3-A *Instruction set architecture (ISA)*, which specifies a weak memory ordering model. With these SoC processors, Apple transitions from Intel-based technology to ARM. Not only does this transition introduce an entirely new ISA, but the ARM architecture also comes with a significantly different memory ordering model [7]. To provide backward compatibility with their former x86-based devices, Apple developed a translation layer called *Rosetta 2*. This translation engine can emulate applications built for x86_64 on Apple Silicon SoCs [8]. Unfortunately, a direct translation on a per-instruction basis alone is insufficient since x86 follows a stricter memory ordering. Every memory access could potentially rely on *Total store ordering (TSO)*. To produce the same behavior as under x86, each access would have to be explicitly synchronized. Instead of paying the accompanying performance costs, Apple implemented TSO directly into their processors. Thus, the M1 SoC has both the ARM and the x86 memory ordering models implemented in hardware, making it the ideal target for comparing these MCMs.

1.1. About this paper

While benchmarks for comparisons between the M1 and other processor families exist [9,10], no research has yet evaluated the performance impact of TSO on M1 SoCs. Additionally, to the best of our knowledge, existing research sparsely conducts evaluations on the *M1 Ultra*, which combines two *M1 Max* dies connected by *UltraFusion*, Apple's custom packaging architecture [11].

In this paper, we evaluate the performance impact of enabling TSO on Apple's M1 Ultra by running synthetic TSO-oriented benchmarks as

* Corresponding author.
E-mail addresses: wrenger@sra.uni-hannover.de (L. Wrenger), toellner@sra.uni-hannover.de (D. Töllner), lohmann@sra.uni-hannover.de (D. Lohmann).

https://doi.org/10.1016/j.sysarc.2024.103102
Received 13 October 2023; Received in revised form 26 January 2024; Accepted 29 February 2024
Available online 4 March 2024

1: X ← 1
2: Y ← 2

(a) CPU0: Store instructions

X	Y
0	0
1	0
1	2

(b) CPU1: Visible with TSO or acq-rel

X	Y
0	0
1	0
0	2
1	2

(c) CPU1: Visible with WO or relaxed

Fig. 1. Observable effect of stores to different memory locations. Given that $X = 0, Y = 0$, each row in Figs. 1(c) and 1(b) represents an observable intermediary state for CPU1, when CPU0 executes the two stores from Fig. 1(a).

well as the CPU benchmarks of SPEC [12]. With our evaluation, we claim the following contributions:

(1) Apple's M1 Ultra benchmark data for the SPEC CPU benchmark suite.
(2) Quantification and analysis of TSO described by the benchmark suite.
(3) Tailor-made synthetic benchmarks for shared memory access times with these MCMs and ARM's atomic instructions.

This paper complements our previous work [13], by taking a deeper look into why certain SPEC CPU benchmarks benefit more from a weaker memory model than others and by expanding the evaluation of the synthetic benchmarks with new core-to-core measurements and additional atomic instructions.

2. Memory consistency models

A *Memory consistency model (MCM)* defines the visible effects of concurrent shared memory access in a distributed system. It is a contract between the developer, the compiler, and the parallel system, providing rules that, if followed, lead to predictable results of shared memory operations. Parallel systems, like x86 or ARM, usually have a relatively lax consistency model for their normal loads and stores and specific instructions to enforce stricter guarantees. With them, they can simulate a stricter MCM if needed.

2.1. Programming model

For hardware independence, most programming languages provide an *atomics* abstraction, such as `std::atomic` in C++ or `std::sync::atomic` in Rust [14,15]. These abstractions define their own MCMs and a set of operations (e.g., `atomic_fetch_add`) that ensure consistency independently from the hardware MCM. The compiler inserts the required instructions and fences to enforce the guarantees where necessary. Usually, *atomics* provide the three memory ordering models listed below in increasing strictness:

relaxed Only loads/stores to the same location are ordered consistently. No guarantees are provided for different memory locations.

acquire-release The acquire-release relation synchronizes accesses to different memory locations for pairs of releasing stores and acquiring loads. All other stores (to different locations) before a releasing store are guaranteed to be visible after an acquiring load of the same location on another processor.

sequential-consistent All sequential-consistent operations are guaranteed to be visible to all processors in the same order.

2.2. Total Store Ordering (TSO) on x86

The x86 architecture guarantees that stores are visible in a consistent order, meaning that each processor observes stores from other processors in the same order [4]. Additionally, every processor also performs stores in program order. Therefore the case that Y is updated before X is impossible, as shown in Fig. 1(b). This ordering is transitive. Other processors observe stores that are causally related in an order consistent with the causality relation. This *Total store ordering (TSO)* already fulfills the *acquire-release* relation for regular loads and stores; thus, no stricter instructions are needed and emitted by the compiler if using the corresponding *atomic* abstractions.

On the downside, the compiled code loses the information of which instructions are expected to be *acquire-release* and which could also be relaxed. This missing information makes it challenging to emulate x86 on systems with weaker memory ordering efficiently, as the optimal placement of fences is an undecidable problem [16]. To provide correctness, x86 emulators (e.g., QEMU) basically insert a fence after every memory instruction.

2.3. Weak Ordering (WO) on ARM

The ARM architecture, on the other hand, has a weak memory ordering model. In the ARMv8 ISA, the concurrency has been revised: In contrast to ARMv7, the architecture now has a *multicopy-atomic model (MCA)*, guaranteeing that modifications to a cache line are linearizable [6]. While this MCM is stricter than the non-MCA ARMv7 model, implementors did not exploit the latter [17]. This multicopy-atomicity guarantees a consistent order of updates to the same location. However, in contrast to x86, stores to different locations are not required to be visible consistently, meaning that every state in Fig. 1(c) can still be observed by other processors. Stronger ordering guarantees can only be enforced with explicit fences or memory barriers (`DMB`, `DSB`) or load, store, compare-and-swap, fetch-add and similar instructions with *acquire-release* semantic (`LDAR`, `STLR`, `LDADDAL`, `CASAL` from ARM A64 [5]). Despite being named *load-acquire* (`LDAR`) and *store-release* (`STLR`), these instructions actually fulfill the sequential-consistent ordering if combined. Consequently, they are relatively slow, as discussed in Section 4.2. Thus, ARMv8.3 introduced `LDAPR`, which allows reordering before `STLR` to different locations and does not make the corresponding `STLR` globally observed [5]. Despite making acquire-release atomics more efficient, this was introduced only recently into compilers: clang added support for `LDAPR` for C/C++ atomics in version 16 (March 2023), GCC in version 13 (April 2023), and Rustc in 1.70 (July 2023). Previously these compilers emitted the stricter `LDAR` for *load-acquire* so that both acquire-release and sequential-consistent loads result in the same assembly.

In general, ARMs laxer memory model gives cores more freedom to reorder instructions, potentially increasing the overall multicore performance for regular (relaxed) instructions. The downside of this is the more complex programming model. Developers have to explicitly synchronize memory accesses if their data structures rely on a specific order of reads and writes. However, this problem might be neglectable, as more and more programming languages have sufficient cross-platform abstractions for *atomics*.

3. The Apple M1 architecture

Apple has disclosed only limited information regarding their custom M1 chips [11,18]. Details on core count, cache and memory sizes, theoretical memory bandwidth, and some performance characteristics have been made public. However, there is no official information about the processor's cache coherence, load and store buffers, micro-operations, instruction schedulers, and execution units. Insights into the microarchitecture stem primarily from reverse engineering projects [19,20].

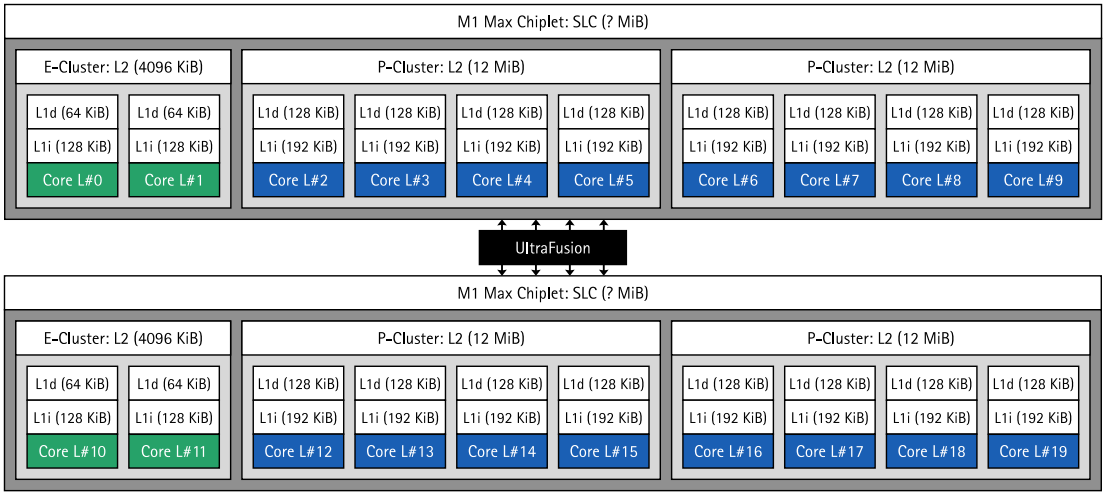


Fig. 2. Cache-Architecture of the M1 Apple Silicon Processor. The E-Clusters contain efficiency "Icestorm" cores, while the P-Clusters consist of performance "Firestorm" cores.

The M1 Ultra SoC consists of two M1 Max chiplets connected through an UltraFusion interconnect, having a reported bandwidth of 2.5 TB/s [11]. A schematic representation of the chiplets and core clusters can be found in Fig. 2. The processor architecture has 16 performance cores grouped in four clusters and four efficiency cores in two clusters. Each processor encompasses separate L1 instruction (L1i) and L1 data (L1d) caches, while an L2 cache is associated with each cluster. Information about a shared last-level (or system-level) cache has not been disclosed. Experimental data indicates that the SLC sizes are 48 MB for the M1 Max and potentially 96 MB for the M1 Ultra [21]. However, this size was not corroborated by our benchmarks. It is also not known if the two SLCs are separated or combined and whether they are shared with the GPU. Regarding cache-line size, `sysctl` on macOS reports a value of 128 B, while `getconf` and the `CTR_EL0` register on Asahi Linux returns 64 B, which is also supported by our measurements.

The M1 Ultra is no conventional ARM processor. It incorporates custom instructions, accelerators, and media units, along with a hardware implementation for TSO, which can be enabled by setting the first bit of the general config register (`ACTLR_EL1`) [20]. After that, normal memory accesses show the same memory ordering behavior as under x86. Unfortunately, further details of this hardware implementation and its limitations are not publicly available.

4. Evaluation

Our test system is an Apple Mac Studio with an M1 Ultra SoC, 128 GiB main memory, and 1 TiB SSD. Our software stack is based on Asahi Linux 6.3.0, a Linux port to Apple Silicon. The TSO memory ordering was toggled on a per-process basis using the recently added `prctl` option `PR_SET_MEM_MODEL_TSO`. The SPEC benchmarks were compiled with GCC 12.1, and the synthetic benchmarks with Rust 1.73.

4.1. CPU benchmarks

To evaluate TSO impact on the M1 Ultra, we choose to run the SPEC CPU 2017 benchmark package [22]. It consists of CPU- and memory-intensive applications, split into 4 benchmark suites with 43 individual benchmarks. SPEC distinguishes between *rate* and *speed* benchmarks, which use different metrics to calculate a system's benchmark score. While the former measures `throughput` of a system, the latter measures `execution time`. A higher benchmark score for the speed benchmarks means less time has been spent on the test system. Additionally, both integrate integer and floating point benchmarks, where the floating point benchmarks make use of heavy parallelism via OpenMP.

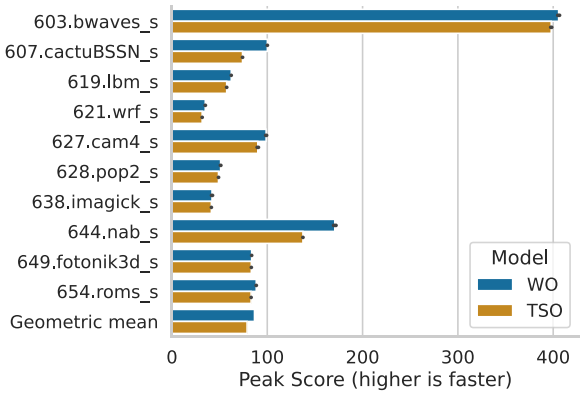


Fig. 3. SPECspeed 2017 parallel floating point benchmarks. Faster execution results in a higher score.

In this evaluation, we focus on the parallel floating point benchmarks since the memory ordering models become significant only with concurrent access to shared memory. The integer benchmarks are single-threaded and, hence, do not use shared memory. Therefore, we cannot expect them to show any performance difference. For the sake of completeness, we confirmed this experimentally but omitted the results in Fig. 3 as they naturally do not provide any value for discussing MCMs. We compiled the benchmarks with the `-Ofast`, `-fprofile-use`, and `-march=native` flags to enable aggressive code optimizations and `-flto` for link-time optimization. During execution, we used 20 threads to utilize all CPU cores of the M1 Ultra. The entire benchmark suite was executed three times and we calculated the median of those runs as the documentation recommends. The final score is calculated by computing the geometric mean of the benchmark medians. This process is executed twice, with and without TSO. However, the benchmark binaries are exactly the same for *Weak ordering (WO)* and TSO.

The results are illustrated in Fig. 3, where the impact of different MCMs varies across individual benchmarks. For instance, in the `649.fotonik3d_s` benchmark, WO achieves a score of 83.64, while TSO records a score of 83.28. Enabling TSO does not affect this benchmark. In contrast, for the `644.nab_s` benchmark, WO scores 170.85, and TSO attains a significantly lower score of 137.43. In the majority of benchmarks, the weak ordering native to the ARMv8 Apple Silicon outperforms TSO. The geometric mean score for the TSO-disabled benchmarks is 86.59, whereas the TSO-enabled benchmarks yield a

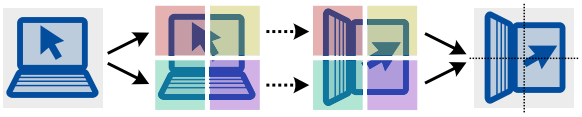


Fig. 4. Basic image transformation in magick. Magic creates distinct image partition copies for each modification thread indicated by different colors. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

geometric mean score of 78.85, translating to a 8.94 percent decrease in performance.

To understand why certain benchmarks profit from weaker memory models while others do not, requires an in-depth investigation of each benchmark. An application can benefit from weak MCMs if it distributes its workload across multiple threads which then access the same memory. Less-optimal access patterns might result in heavy cache-line bouncing between cores. In a weak MCM, cores can reschedule their instructions more effectively to hide cache misses while stronger MCMs might have to stall more frequently. In contrast, applications that rarely operate on shared memory, also profit hardly from weaker memory models. Since all our SPEC benchmarks schedule their workloads across threads, we chose to analyze how they access shared memory. Particularly, we manually analyzed the source code of `644.nab_s` and `638.imagick_s` because the former benefits significantly from WO while the latter does not.

The `638.imagick_s` benchmark applies various transformations to input images. The general idea is visualized in Fig. 4 where thread-private data is highlighted in different colors. For the process of a simple 90-degree rotation, `magick` divides the image into distinct non-overlapping regions that are transferred to the worker threads. They can then apply their transformations locally without the need for synchronization. After the transformation, the thread copies its chunk to the original image in parallel. This also requires no synchronization, because these regions do not overlap and the threads do not overwrite each other's data. This approach allows for fast image manipulation, but since threads mostly do not operate on shared memory `magick` does not profit from a weaker memory ordering model. There may be a small amount of cache-line bouncing on the boundaries of the regions, but these effects are neglectable as shown by our results.

The `644.nab_s` benchmark consists of parallel floating point calculations for molecular modeling. In contrast to `magick`, the authors of this benchmark specifically optimized their data structures for the cache hierarchy. Each worker thread operates on 64-byte chunks, aiming to eliminate false sharing. However, only resizing the data structures to cache lines is not enough. These chunks must also be aligned to the start of a cache-line to really prevent false sharing. If not properly aligned, two cores still share the same cache-line as these chunks span over two instead of one cache-line. As shown in Fig. 5, the consequence is an enormous cache-line pressure where one cache-line is permanently bouncing between two cores. This high pressure can enforce stalls on architectures with stronger MCMs like TSO, that wait until a core can exclusively claim a cache-line for writing, while weaker memory models are able to reschedule instructions more effectively. Consequently, `644.nab_s` performs 24.32 percent better under WO compared to TSO.

4.2. Synthetic benchmarks

We devised two synthetic benchmarks to delve deeper into the performance discrepancies observed in the SPECS benchmarks: (1) a *store* benchmark and (2) a *ldadd* benchmark. Both benchmarks employ a shared memory buffer between two threads: a writer, responsible for updating the buffer, and a reader, tasked with observing these updates. The benchmarks vary only in the instruction utilized for the

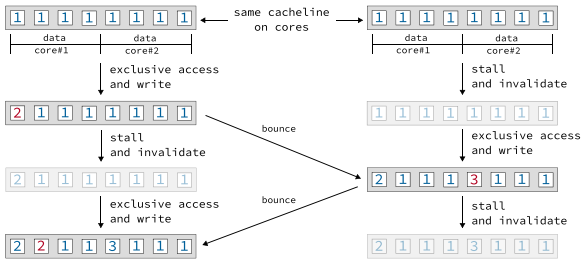


Fig. 5. Cacheline bouncing for misaligned data.

buffer updates: The writer thread iterates through the buffer in 64-byte (cache-line) steps, executing either *stores* or *ldadds* to increment the numbers within the first 8 bytes of each element as shown in Fig. 6. Initially, all elements are zero, and in the first iteration, they are all incremented to one, then in the second iteration to two, and so forth. The *store* benchmark uses a store operation to write the number of the current iteration (from a register) to all elements, while the *ldadd* benchmark uses this instruction to increment the previous values, resulting in the same general behavior.

Concurrently, the reader iterates through the buffer, loading and comparing pairs of adjacent elements. It observes out-of-order updates where the second element is larger than the first, indicating that the stores/ldadds were perceived in a different order from the writer's execution. This phenomenon only occurred under weak ordering; when TSO was enabled, no out-of-order updates were detected. Apart from the shared buffer and a boolean value for synchronizing the beginning and end of the measurement, the threads do not access any shared data. They also do not synchronize between iterations; thus, the reader usually finishes more iterations than the writer.

In these benchmarks, we counted the number of iterations each thread could complete within one second. This value was then multiplied by the buffer length to calculate the operations per second. The benchmarks were compiled with relaxed instructions (LDR and STR or LDADD) and the exact same binaries were executed with and without TSO enabled. The reader and writer threads were pinned to different cores of either the same cluster, sharing an L2 cache, a separate cluster on the same chiplet, or different chiplets.

Store benchmark. Fig. 7(a) depicts the number of parallel stores and loads for every core combination on a shared 1 MiB buffer. The first row shows the number of stores, executed by the writer cores under WO and TSO. The second row documents the number of loads that were executed at the same time on the corresponding reader core. When focusing on the performance cores (2–9 and 12–19) of the second row for TSO, we clearly see three different performance classes: Cores of (1) the same group as blue squares, (2) different groups on the same chiplet as yellow squares, and (3) different groups on different chiplet as red squares. These three performance classes directly match the architecture of the M1 Ultra (Section 3). The best performance is achieved when the reader and writer threads are pinned to the same core groups. Here the cores share the same L2 cache, which (with its 12 MiB) is large enough to contain the whole buffer. Contrary, we see the worst performance for cores on different chiplets.

Fig. 7(b) shows the number of parallel stores for these three classes on different buffer sizes. The horizontal lines indicate the cache sizes (128 KiB, 12 MiB and 96 MiB as described in Section 3). Looking at the store performance (upper row), we see that it is relatively low for buffers that fit in the L1 cache, likely due to constant cache invalidations. Meanwhile, for buffers with sizes between the L1 and L2 cache, the highest number of stores occurs on the same cluster. This performance drops significantly on different clusters where the L2 cache is not shared. For buffers larger than the L2 cache, the performance is similar regardless of the cores used. The limits of the

```rust
while !running.load(Relaxed) {} // wait

let mut iter = 0;
while running.load(Relaxed) {
    for item in buffer {
        item.store(iter as _, Relaxed);
    }
    iter += 1;
}
```

(a) Writer thread of the *store* benchmark.

```rust
while !running.load(Relaxed) {} // wait

let mut iter = 0;
while running.load(Relaxed) {
    for item in buffer {
        item.fetch_add(1, Relaxed);
    }
    iter += 1;
}
```

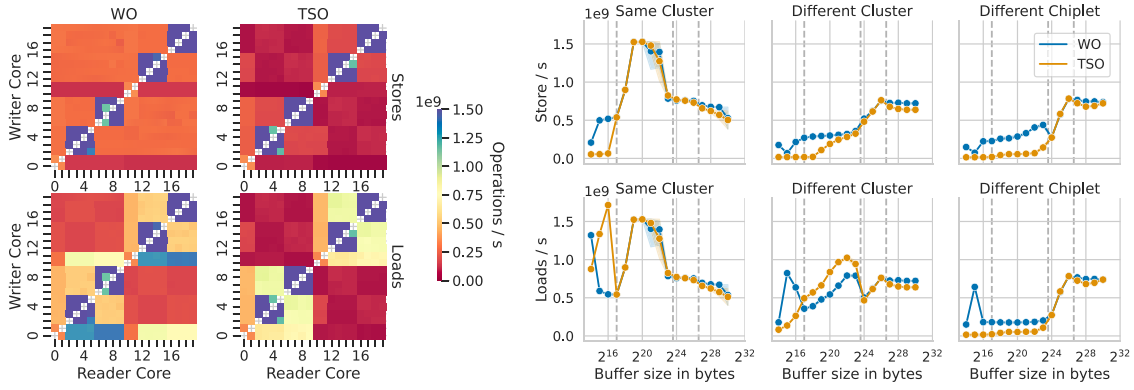
(b) Writer thread of the *ldadd* benchmark.

```rust
while !running.load(Relaxed) {} // wait

let mut iter = 0;
let mut ooo = 0; // out-of-order counter
while running.load(Relaxed) {
    for pair in buffer.windows(2) { // sliding window
        ooo += (pair[0].load(Relaxed) < pair[1].load(Relaxed)) as usize;
    }
    iter += 1;
}
```

(c) Reader thread of both benchmarks.

Fig. 6. Source code snippets of the reader and writer threads, with the `buffer` containing cache-line aligned atomic integers.



(a) Core-to-core performance on a 1 MiB buffer.

(b) Performance for different buffers sizes.

Fig. 7. Concurrent store (top) and load (bottom) operations.

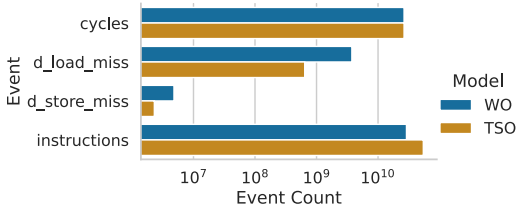


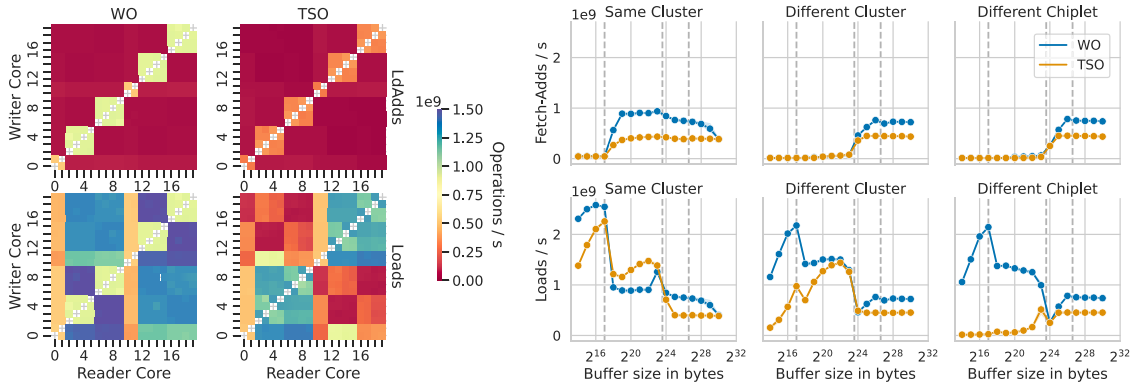
Fig. 8. Perf counters for the *store* benchmark, executed on the same cluster with a 2^{16} bytes (64 KiB) buffer.

L1 and L2 cache sizes are clearly visible, while the SLC is not so apparent. We only observe that the performance stops increasing for buffers larger than 96 MiB (the SLC size).

The read performance (bottom row), with TSO enabled, is higher for buffers smaller than the L1 cache. This seems to be a pattern when comparing weak stores and loads on small buffers: The lower the store performance is, the faster loads tend to become. This inverse effect might be attributed to fewer cache invalidations, as TSO writes are considerably slower. The performance counters, shown in Fig. 8, support this observation: The number of load and store misses is higher on weak ordering, where the number of writes is also significantly higher. For buffers between the L1 and L2 cache sizes, the highest number of loads occurs on the same cluster. The performance drop is not as significant for different clusters on the same chiplet but is more pronounced between chiplets. Also, TSO loads are slightly faster for L2-sized buffers on different clusters. For buffers larger than the L2 cache, the performance is again very similar across different configurations.

Ldadd benchmark. The second synthetic benchmark uses LDADD instructions in the writer thread to increment the buffer elements. When comparing the *ldadd* benchmark (Fig. 9) with the *store* benchmark (Fig. 7), we see that ldadds (first row) are, at best, only half as fast as stores. Also, enabling TSO decreases the ldadd performance even further. Weakly-ordered ldadds are up to twice as fast as their TSO counterpart, especially on the same cluster for buffers between the L1 and L2 cache sizes. Again, the instructions are far slower for L1- and L2-sized buffers on different clusters. However, this difference is even more pronounced compared to the *store* benchmark.

The load performance also changed significantly from the *store* benchmark. With TSO enabled, this time, the read performance is slower for small buffers but faster for buffers between the L1 and L2 cache sizes on the same cluster. On different clusters, TSO reads are now consistently slower than weakly ordered ones. We again see that lower ldadd performance generally results in higher load performance, possibly again due to the lower cache miss rate.



(a) Core-to-core performance on a 1 MiB buffer. (b) Performance for different buffers sizes.

Fig. 9. Concurrent ldadd (top) and load (bottom) operations.

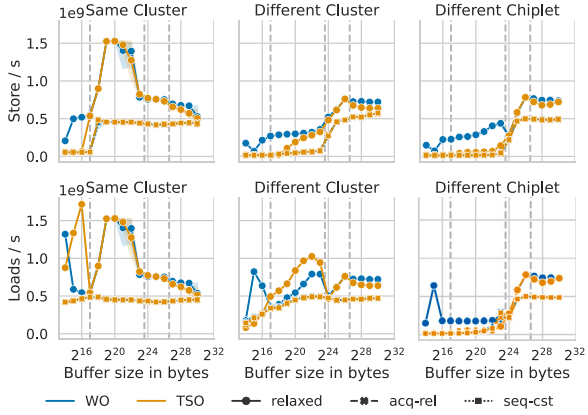


Fig. 10. Concurrent store and load operations, comparing different atomic instructions.

Stricter atomic instructions. The ARMv8.3 architecture also provides instructions with stricter memory ordering guarantees, namely STLR, LDADDAL, LDAR, and LDAPR (Section 2.3). These instructions are emitted by compilers for the acquire-release and sequential-consistent atomic operations. For completeness, we also evaluated the performance of these instructions in our synthetic benchmarks. As illustrated in Fig. 10, we see the performance of these atomic orderings compared with the standard relaxed instructions (STR, LDR). The stricter instructions are consistently slower than the relaxed instructions and their performance is independent of WO and TSO. The latter is to be expected as they enable even less instruction reordering than TSO. This performance gap also might give insight into Apple's decision to fully support TSO in their SOCs. Notable is also the acquire-release operations do *not* show improved performance on the Apple M1, even with newer compilers emitting the supposedly more optimized LDAPR instructions. These trends are similar in the *ldadd* benchmark. In summary, our analysis of the *store* and *ldadd* benchmarks reveals several performance nuances based on buffer sizes and the relationship between the reader and writer threads. We see that stores and ldadds are generally and sometimes drastically slower under TSO. With a few exceptions, the load performance also seems to be faster on weak ordering.

5. Discussion

The measurable effects of different types of Memory consistency models highly depend on the access patterns to shared memory as well as the system's cache hierarchy. Looking back at Section 4.1, we see that the impact of different MCMs on the individual benchmark fluctuates. Without more detailed information about the inner workings of the M1 architecture, its microarchitecture, and cache hierarchy, we can only speculate on the reasons for these performance variations: The primary performance advantage applications might gain from running under weaker memory ordering models like WO is due to greater instruction reordering capabilities. Therefore, the performance benefit vanishes if the hardware architecture cannot sufficiently reorder the instructions (e.g., due to data dependencies).

Furthermore, the synthetic benchmarks suggest that the performance difference highly depends on the size of the application's working set and the cores accessing the shared memory. The write (store, ldadd) performance is consistently higher on weak ordering. However, the load performance might be faster under TSO when the corresponding write performance is very low, and consequently, fewer cache invalidations happen. Instructions with acquire-release semantics are significantly slower on the M1 than both WO and TSO, which explains Apple's decision to implement TSO for the x86 emulation instead of relying on these instructions. Unfortunately, the cache implementation is not public, which limits the interpretability of these results.

Strict models like sequential consistency prohibit hardware from reordering instructions but make it easier for developers and compilers to reason about parallel code. Or, from another perspective, the freedom of hardware reordering instructions *requires* developers and compilers to thoroughly reason about the order in which the emitted code is executed to ensure the program's semantics remain correct. In this setting, the novel feature of the Apple M1, where the MCM is configurable at run time, provides interesting flexibility for software developers and compilers.

6. Related work

The field of memory consistency models has been under active research for a couple of decades. With the emergence of multiprocessor systems, the sequentiality properties of those systems needed to be properly formalized. In a seminal paper from 1979, Lamport describes sequential consistency as the property of a multiprocessor system to run all instructions of all processors in some sequential order and that each processor strictly follows its program instruction order [23]. Instruction reordering, however, can provide a considerable performance benefit if the CPU can reschedule instructions to reach a higher cache hit rate. Therefore, over the following years, many different other consistency models have been established, such as WO [2], processor consistency [24], partial store ordering, TSO, and many others. While most hardware commonly follows a specific consistency model, there are a few systems in the wild next to the M1 that allow toggling between different MCMs dynamically, during runtime and in hardware. Notably, all architectures that include a SPARC v8 Reference MMU implementation allow to switch between PSO and TSO during runtime by toggling the PSO bit in the MMU control register of a

specific processor [25]. The key difference between SPARC systems and the M1 is that the latter can switch to WO as an alternative MCM, which is more relaxed compared to PSO and therefore allows further instruction reordering. With the new release of SPARC v9, the successor to SPARC v8, a new in-hardware toggleable MCM has been added: relaxed ordering [26]. Relaxed ordering under SPARC v9 is even closer to WO on ARM compared to parallel store ordering, as it allows further instruction rescheduling. SPARC v9 systems and ARM, however, provide different synchronization primitives if instruction rescheduling needs to be prohibited. While the former provides more coarse-grained, global synchronization primitives, ARM comes with smaller, distinct shareability domains to limit the necessity of synchronization.

To investigate the performance impact of these consistency models, several benchmarks have been conducted. Gharachorloo et al. [27] measured the effect of different MCMs on a simulated Stanford DASH multiprocessor architecture. Their results have shown that stricter ordering models performed significantly worse than less strict models for architectures with blocking reads. A more recent study by Naeem et al. [28] draws the same conclusion on network-on-chip-based distributed shared memory systems, improving their system performance when transitioning from stricter to weaker memory consistency models.

Moving from stricter to weaker models shifts the responsibility of sequentiality from the hardware to the software and software toolchain. This inherently enforces research on how to express program sequentiality as a developer and how to emit appropriate instructions as a compiler. In the paper of Boehm et al. [29], the authors describe a divergence between C/C++ being single-threaded programming languages while giving additional multithread support via an additional library. Since the language itself does not provide intrinsic support for multithreaded code, it is up to the libraries to offer synchronization primitives for concurrent access to shared resources, such as a shared address space, that enforce a specific order for particular instructions. Enforcing a specific order is achieved by properly placing memory barriers, guaranteeing that certain load/store operations execute before/after surrounding instructions. Shaked et al. [30] investigate the impact of memory barriers on mixed-size memory accesses of different data widths. Today's processors commonly allow accessing memory at granularities of 1, 2, 4, or 8 bytes. Placing barriers for mixed use of those granularities should enforce the same ordering as for data accesses of equal width. This general assumption, however, proves to be wrong for ARMv8 and POWER architectures, as the authors' evaluation clarifies. While placing a strong memory barrier between every memory access of equal width for architectures implementing WO results in a sequential-consistent behavior, this is not the case for mixed-size memory accesses.

Other research regarding Apple's M1 processors is sparse. [9] benchmarked the M1 and M1 Ultra for high-performance scientific computing and compared its GPU performance against two Nvidia-equipped servers, while [10] studied their energy efficiency. ARM systems, in general, have been evaluated against x86 systems on different, primarily HPC-based workloads [31–33]. Kodama et al. [34] evaluated the performance of the ARM A64FX against a dual-socket Xeon using the SPEC CPU and OMP benchmarks. Nevertheless, none of these works focused specifically on memory-ordering differences.

7. Conclusion

The Apple M1 is the first processor that implements both, ARM's weak memory ordering and Intel's TSO, as a software-configurable feature. This also makes it possible for the first time to compare the performance impact of the different memory models on real hard- and software.

Our results show a significant difference in the multicore performance when comparing both models. Despite being more challenging to program for, the weak model is generally faster: 8.94 percent on average running SPEC CPU and more than twice as fast in some of our synthetic benchmarks. However, the lack of knowledge about the internals of the M1 architecture makes it hard to fully explain all effects of TSO on this SoC. Our results suggest that these are deeply entangled with the caching hierarchy and memory access path. Nonetheless, we think that this work is an essential step toward understanding the actual runtime effects of the memory ordering models.

CRediT authorship contribution statement

Lars Wrenger: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Visualization, Writing – original draft, Writing – review & editing. **Dominik Töllner:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, Writing – original draft, Writing – review & editing. **Daniel Lohmann:** Conceptualization, Funding acquisition, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank our reviewers for their valuable feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – LO 1719/8-1.

References

[1] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, SIGARCH Comput. Archit. News 18 (2SI) (1990) 15–26, http://dx.doi.org/10.1145/325096.325102.

[2] M. Dubois, C. Scheurich, F. Briggs, Memory access buffering in multiprocessors, in: Proceedings of the 13th Annual International Symposium on Computer Architecture, ISCA '86, IEEE Computer Society Press, Washington, DC, USA, 1986, pp. 434–442.

[3] L. Higham, J. Kawash, N. Verwaal, Defining and Comparing Memory Consistency Models, University of Calgary, 1997.

[4] Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, 2022, Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. (Accessed 30 May 2023).

[5] ARM Cortex-A Series – Programmer's Guide for ARMv8-A, ARM Limited, 2015.

[6] Learn the Architecture – Memory Systems, Ordering, and Barriers, ARM Limited, 2022, https://developer.arm.com/documentation/102336/0100. (Accessed 30 May 2023).

[7] Apple announces Mac transition to Apple silicon, 2020, https://nr.apple.com/d2O2Y718J3. (Accessed 22 March 2023).

[8] Rosetta translation environment, 2023, https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment. (Accessed 22 March 2023).

[9] C. Kenyon, C. Capano, Apple silicon performance in scientific computing, in: 2022 IEEE High Performance Extreme Computing Conference, HPEC, 2022, pp. 1–10, http://dx.doi.org/10.1109/HPEC55821.2022.9926315.

[10] Z. Ali, T. Tanveer, S. Aziz, M. Usman, A. Azam, Reassessing the performance of ARM vs x86 with recent technological shift of apple, in: 2022 International Conference on IT and Industrial Technologies, ICIT, 2022, pp. 01–06, http://dx.doi.org/10.1109/ICIT56493.2022.9988933.

[11] Apple M1 ultra, 2022, https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/. (Accessed 22 March 2023).

[12] The standard performance evaluation corporation, 2023, https://www.spec.org/ (Accessed 22 March 2023).

[13] L. Wrenger, D. Töllner, D. Lohmann, TOSTING: Investigating total store ordering on ARM, in: Proceedings of the 36th GI/ITG International Conference on Architecture of Computing Systems, ARCS 23, Springer International Publishing, Athens, Greece, 2023, http://dx.doi.org/10.1007/978-3-031-42785-5_10.

[14] C++ Atomic operations library, 2023, https://en.cppreference.com/w/cpp/atomic (Accessed 26 March 2023).

[15] Rust Standard Library – Module std::sync::atomic, 2023, https://doc.rust-lang.org/std/sync/atomic/index.html. (Accessed 26 March 2023).

[16] M.F. Atig, A. Bouajjani, S. Burckhardt, M. Musuvathi, What's decidable about weak memory models? in: H. Seidl (Ed.), ESOP, in: Lecture Notes in Computer Science, Springer-Verlag, 2021, pp. 26–46.
[17] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, P. Sewell, Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8, Proc. ACM Program. Lang. 2 (POPL) (2017) http://dx.doi.org/10.1145/3158107.
[18] M. Mattioli, Meet the FaM1ly, IEEE Micro 42 (3) (2022) 78–84, http://dx.doi.org/10.1109/MM.2022.3169245.
[19] D. Johnson, Apple M1 microarchitecture research, 2023, https://dougallj.github.io/applecpu/firestorm.html. (Accessed 22 March 2023).
[20] Asahi linux wiki, 2023, https://github.com/AsahiLinux/docs/wiki. (Accessed 22 March 2023).
[21] Apple's M1 pro, M1 max SoCs investigated: New performance and efficiency heights, 2021, https://www.anandtech.com/show/17024/apple-m1-max-performance-review. (Accessed 22 March 2023).
[22] SPEC CPU benchmark package, 2023, https://www.spec.org/cpu2017/ (Accessed 27 March 2023).
[23] Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. C-28 (9) (1979) 690–691, http://dx.doi.org/10.1109/TC.1979.1675439.
[24] J.R. Goodman, Cache Consistency and Sequential Consistency, University of Wisconsin-Madison Department of Computer Sciences, 1991, http://digital.library.wisc.edu/1793/59442.
[25] C. SPARC International, Inc., The SPARC Architecture Manual: Version 8, Prentice-Hall, Inc., USA, 1992.
[26] C. SPARC International, Inc., The SPARC Architecture Manual: Version 9, Prentice-Hall, Inc., USA, 1994.
[27] K. Gharachorloo, A. Gupta, J. Hennessy, Performance evaluation of memory consistency models for shared-memory multiprocessors, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS IV, Association for Computing Machinery, New York, NY, USA, 1991, pp. 245–257, http://dx.doi.org/10.1145/106972.106997.
[28] A. Naeem, X. Chen, Z. Lu, A. Jantsch, Realization and performance comparison of sequential and weak memory consistency models in network-on-chip based multi-core systems, in: 16th Asia and South Pacific Design Automation Conference, ASP-DAC 2011, 2011, pp. 154–159, http://dx.doi.org/10.1109/ASPDAC.2011.5722176.
[29] H.-J. Boehm, S.V. Adve, Foundations of the C++ concurrency memory model, in: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 68–78, http://dx.doi.org/10.1145/1375581.1375591.
[30] S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K.E. Gray, A. Sezgin, M. Batty, P. Sewell, Mixed-size concurrency: ARM, POWER, C/C++11, and SC, in: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 429–442, http://dx.doi.org/10.1145/3009837.3009839.
[31] N. Gupta, R. Ashiwal, B. Brank, S.K. Peddoju, D. Pleiter, Performance evaluation of ParalleX execution model on arm-based platforms, in: 2020 IEEE International Conference on Cluster Computing, CLUSTER, 2020, pp. 567–575, http://dx.doi.org/10.1109/CLUSTER49012.2020.00080.
[32] P. Ouro, U. Lopez-Novoa, M.F. Guest, On the performance of a highly-scalable Computational Fluid Dynamics code on AMD, ARM and Intel processor-based HPC systems, Comput. Phys. Comm. 269 (2021) 108105, http://dx.doi.org/10.1016/j.cpc.2021.108105.
[33] J. Xia, C. Cheng, X. Zhou, Y. Hu, P. Chun, Kunpeng 920: The first 7-nm chiplet-based 64-core ARM SoC for cloud services, IEEE Micro 41 (5) (2021) 67–75, http://dx.doi.org/10.1109/MM.2021.3085578.
[34] Y. Kodama, M. Kondo, M. Sato, Evaluation of SPEC CPU and SPEC OMP on the A64FX, in: 2021 IEEE International Conference on Cluster Computing, CLUSTER, 2021, pp. 553–561, http://dx.doi.org/10.1109/Cluster48925.2021.00088.