TOSTING: Investigating Total Store Ordering on ARM

Lars Wrenger, Dominik Töllner, and Daniel Lohmann

Systems Research and Architecture Group, Leibniz Universität Hannover, Germany
{wrenger,toellner,lohmann}@sra.uni-hannover.de

Abstract. The Apple M1 ARM processors incorporate two memory consistency models: the conventional ARM weak memory ordering and the *total store ordering (TSO)* model from the x86 architecture employed 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 various workloads, 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.

Keywords: TSO · Memory Ordering · Apple M1

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) in shared-memory systems 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 [20,17,23]. These models differ in how strict guarantees they provide. Both x86 and ARM define a MCM that allows (limited) reordering of instructions [5,1,6]. 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 and potentially reduce synchronization overheads

between caches. Seeing this tradeoff between higher performance and simpler programming models, we ask how extensive the performance benefits really 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. Together with introducing an entirely new ISA, these Apple Silicon SoCs also significantly change the memory model the hardware now operates on [2]. 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 [9]. 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 built 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 [25,14], 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 [4].

In this paper, we evaluate the performance impact of enabling TSO on Apple's M1 Ultra by running synthetic TSO-oriented benchmarks as well as the CPU benchmarks of `SPEC`, a non-profit corporation to establish standardized benchmarks [12]. With our evaluation, we claim the following contributions:

(1) Apple's M1 Ultra benchmark data for the SPEC CPU benchmark suite.
(2) Quantification of TSO described by the benchmark suite and tailor-made synthetic test cases.

2 Memory Consistency Models

The *memory consistency model (MCM)* defines the correct behavior of shared memory for concurrent access. 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 [8,10]. 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 memory) before a releasing store are guaranteed to be visible after an acquiring load of the same memory 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 on x86

1: X ← 1
2: Y ← 2

(a) CPU0: Store instructions

X	Y
0	0
1	0
1	2

(b) CPU1: Visibility with TSO or acquire-release

X	Y
0	0
1	0
0	2
1	2

(c) CPU1: Visibility with weak/relaxed ordering

Fig. 1: Observable effect of stores to different memory locations.

Given that $X = 0, Y = 0$, each row in Fig. 1c and Fig. 1b represents an observable intermediary state for CPU1, when CPU0 executes the two stores from Fig. 1a.

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 [5]. Additionally, every processor also performs stores in program order. Therefore the case that Y is updated before X is impossible, as shown in Fig. 1b. 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 [15]. To provide correctness, x86 emulators (e.g., QEMU) basically insert a fence after every memory instruction.

2.3 Weak Ordering 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 [31]. 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. 1c can still be observed by other processors (CPU1). 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 [1]). Despite being named *load-acquire* (`LDAR`) and *store-release* (`STLR`), these instructions actually fulfill the sequential-consistent memory ordering if combined. Consequently, they are relatively slow, as discussed in Sec. 4.2. Thus, ARMv8.3 introduced `LDAPR`, which allows reordering before `STLR` to different locations [1]. Despite making acquire-release atomics more efficient, `LDAPR` is still not used by most compilers for *load-acquire* (instead `LDAR` is emitted). Recently, clang added support for `LDAPR` in C/C++ atomics in version 16 (March 2023), GCC in version 13 (April 2023), and for Rust, this is still only available on the nightly channel.

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 might rely on the order of writes. However, this might not be a problem, 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 [4,28]. Details on core counts, 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 [24,7].

The M1 Ultra *system on a chip (SoC)* consists of two M1 Max chiplets connected through an UltraFusion interconnect, having a reported bandwidth

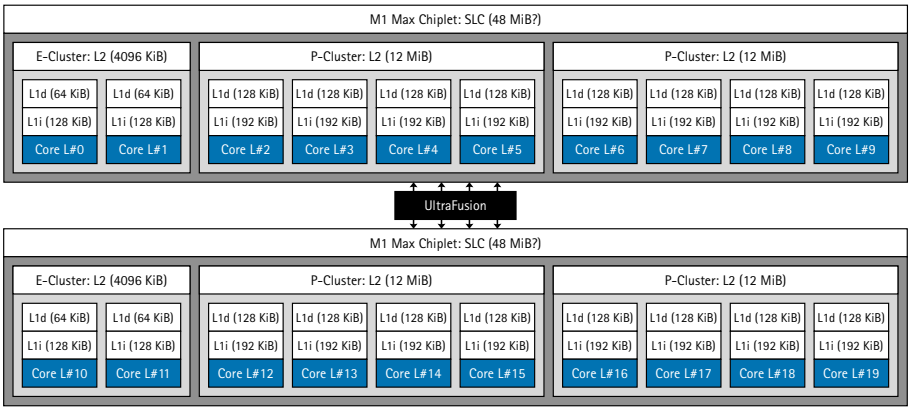

Fig. 2: Cache-Architecture of the M1 Apple Silicon Processor
The E-Clusters each contain two efficiency cores (codename "Icestorm"), while the P-Clusters consist of four performance cores (codename "Firestorm"). Each core has L1 data and instruction caches and shares the L2 cache with the rest of the cluster.

of 2.5TB/s [4]. 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 48MB for the M1 Max and potentially 96MB for the M1 Ultra [3]. However, these values were not corroborated by our benchmarks. It is also not known if the two SLCs are separated or combined. Regarding cache-line size, `sysctl` on macOS reports a value of 128 B, while `getconf` and the `CTR_EL0` register on Asahi Linux return 64 B, which is also supported by our measurements.

The M1 Ultra is not a 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`) [7]. 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 publically 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 6.1.0, a Linux port to Apple Silicon. The TSO memory ordering was toggled system-wide for all cores using a kernel module [13] before executing the respective benchmark. The SPEC benchmarks were compiled with GCC 12.1, and the synthetic benchmarks with Rust 1.69.0.

4.1 CPU Benchmarks

To evaluate TSO impact on the M1 Ultra, we choose to run the `SPEC CPU 2017` benchmark package [11]. This package consists of 4 benchmark suites with 43 individual benchmarks. SPEC generally 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 speed benchmarks means less time has been spent on the *system under test (SUT)* (here, the M1 Ultra). Additionally, both integrate integer and floating point benchmarks, where especially the floating point benchmarks make use of heavy parallelism via OpenMP.

In this evaluation, we focus on the *SPECspeed 2017 Floating Point* suite since the utilization of heavy parallelism results in many hardware threads accessing shared memory concurrently, allowing us to evaluate different memory ordering models properly. We utilize all CPU cores within the M1 Ultra, resulting in a total of 20 threads in execution for every benchmark issued. The benchmarks run CPU- and memory-intensive code such as 3D simulation, modeling of physical systems and their behavior, as well as image manipulation. We execute three iterations of the floating point benchmark suite and select the median of those iterations as the documentation recommends. A final score is calculated by computing the geometric mean of all selected medians of all benchmarks. While the suite provides two benchmark tuning modes *base* and *peak*, we only show the peak version that uses more platform-specific optimizations in this paper. However, the base configuration exhibits similar trends. This whole suite is executed twice, once for enabled TSO and once for disabled TSO. The benchmark code does not contain any atomic operations, hence neither the compiler nor the hardware are hinted to emit/execute such instructions. Therefore, the application binary code is 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 83.63, while TSO records a score of 83.27. Enabling TSO does not affect this benchmark. In contrast, for the `644.nab_s` benchmark, WO scores 171.34, 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 86.57, whereas the TSO-enabled benchmarks yield a geometric mean score of 78.83, translating to a 8.94 percent decrease in performance.

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 *fetch-add* 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

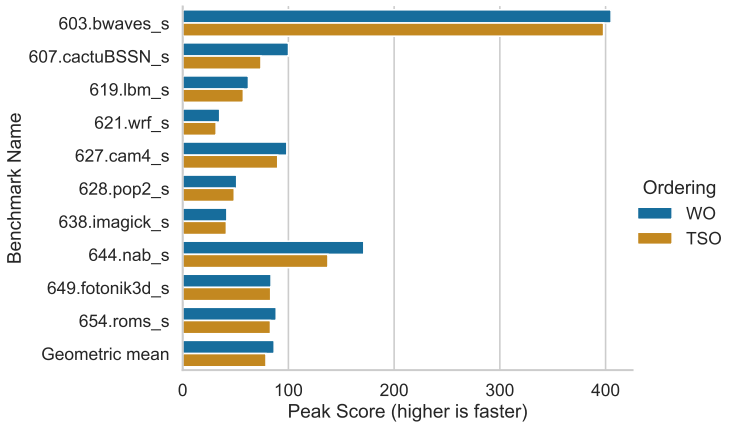

Fig. 3: SPECspeed 2017 Floating Point
Comparison of the parallel SPEC CPU benchmarks. Faster execution results in a higher score.

utilized for buffer updates: The writer thread iterates through the buffer in 64-byte (cache-line) steps, executing either *stores* or *fetch-adds* to increment the numbers within the first 8 bytes of each element. 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 (1) uses a store operation to write the current iteration to all elements, while the *fetch-add* benchmark (2) 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 and counts out-of-order updates where the second element is smaller than the first, indicating that the update operations 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 utilized 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 (`LDR` and `STR` or `LDADD`) and acquire-release (`LDAR` and `STLR` or `LDADDAL`) instructions. These exact same binaries were then 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.

Regarding the *store* benchmark, Fig. 4 shows the number of parallel stores (Fig. 4a) and loads (Fig. 4b) for varying buffer sizes. The horizontal lines indicate

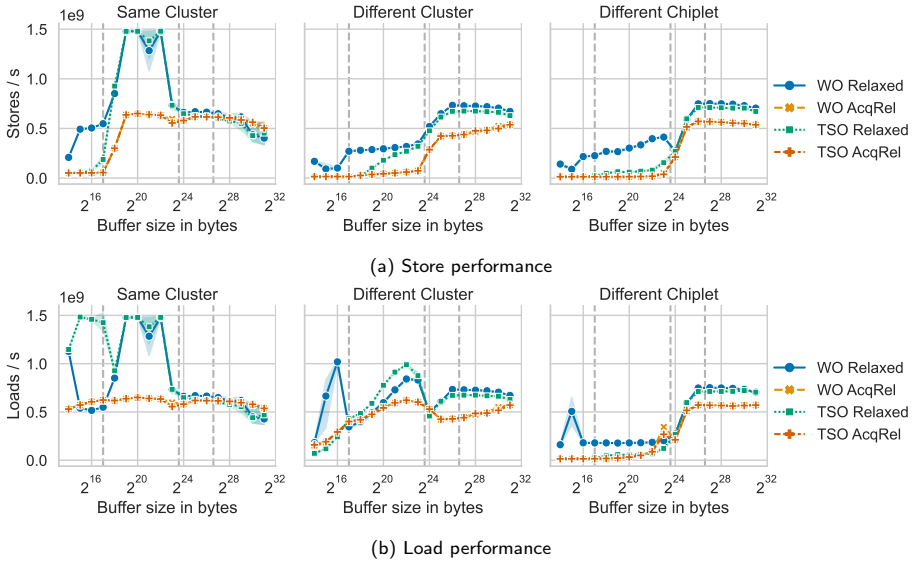

Fig. 4: Concurrent store and load operations
The writer (top) and reader (bottom) threads were pinned to different cores of the same cluster (left), separate clusters of the same chiplet (middle), and different chiplets (right). The gray horizontal lines mark the cache sizes (L1 = 128 KiB, L2 = 12 MiB and SLC = 96 MiB).

the cache sizes (128 KiB, 12 MiB and 96 MiB as described in Sec. 3). Our first observation is that, for all benchmarks, enabling TSO does not impact the performance of the acquire-release instructions. This outcome is to be expected, as these acquire-release instructions employ an explicit and even stricter sequentially-consistent memory ordering (Sec. 2.3), making them generally slower than weak ordering and TSO.

Looking at the store performance of the first benchmark, we see that it is pretty low for buffers that fit in the L1 cache, possibly due to cache invalidations (Fig. 4a). 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 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, with TSO enabled, is faster for buffers smaller than the L1 cache (Fig. 4b). 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. 5, support this observation: The number of load and store misses is higher on weak

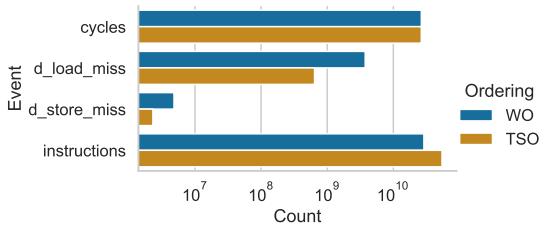

Fig. 5: Perf counters for the *store* benchmark
The benchmark was executed on the same cluster with a 2^{16} bytes buffer. The events were measured for both the writer and reader threads together.

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.

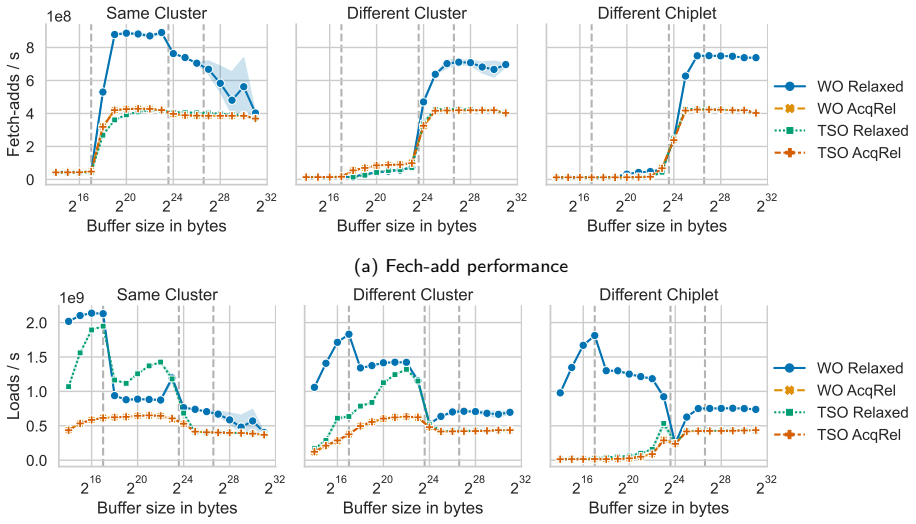

Fig. 6: Concurrent fetch-add and load operations

The second synthetic benchmark used fetch-adds (`LDADD` / `LDADDAL`) in the writer thread to increment the buffer elements (Fig. 6). When comparing the *ldadd* benchmark (Fig. 6a) with the *store* benchmark (Fig. 4a), we see that fetch-adds are, at best, only half as fast as stores. Also enabling, TSO decreases

the fetch-add performance to or even below the acquire-release instructions. This differs from the previous benchmark, where the TSO stores were generally above their acquire-release counterparts. Meanwhile, weakly-ordered fetch-adds are almost twice as fast, especially for buffers between the L1 and L2 cache sizes with the reader and writer on the same cluster. Again, the instructions are far slower for L1-sized buffers and L2-sized buffers on different clusters. However, this difference is even more pronounced compared to stores.

The load performance (Fig. 6b) 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 fetch-add performance generally results in higher load performance.

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 fetch-adds 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 of different actors of a shared memory system as well as its cache hierarchy. Looking back at Sec. 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, fetch-add) 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. Fully understanding these variations requires a more in-depth examination of the cache implementation.

Strict models like *sequential consistency (SC)* 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 [27]. 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 [17], *processor consistency (PC)* [21], *partial store ordering (PSO)*, 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 [32]. 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 (RO)* [33]. RO under SPARC v9 is even closer to WO on ARM compared to PSO, 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. [19] 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. [29] draws the same conclusion on *network-on-chip (NOC)* based distributed shared memory multicore 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. [16], 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. [18] 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. [25] benchmarked the M1 and M1 Ultra for high-performance scientific computing and compared its GPU performance against two Nvidia-equipped servers, while [14] studied their energy efficiency. ARM systems, in general, have been evaluated against x86 systems on different, primarily HPC-based workloads [22,30,34]. Kodama et al. [26] 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.

In our results, we see a significant effect on 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.

References

1. ARM Cortex-A Series – Programmer's Guide for ARMv8-A. Arm Limited (mar 2015)
2. Apple announces Mac transition to Apple silicon (2020), https://nr.apple.com/d2O2Y718J3 – accessed 2023-03-22
3. 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 2023-03-23
4. 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 2023-03-22
5. Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Intel (2022), https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html – accessed 2023-05-30
6. Learn the architecture – Memory Systems, Ordering, and Barriers. Arm Limited (jun 2022), https://developer.arm.com/documentation/102336/0100 – accessed 2023-05-30
7. Asahi linux wiki (2023), https://github.com/AsahiLinux/docs/wiki – accessed 2023-03-23
8. C++ atomic operations library (2023), https://en.cppreference.com/w/cpp/atomic – accessed 2023-03-26
9. Rosetta Translation Environment (2023), https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment – accessed 2023-03-22
10. Rust standard library – module std::sync::atomic (2023), https://doc.rust-lang.org/std/sync/atomic/index.html – accessed 2023-03-26
11. SPEC CPU benchmark package (2023), https://www.spec.org/cpu2017/ – accessed 2023-03-27
12. The Standard Performance Evaluation Corporation (2023), https://www.spec.org/ – accessed 2023-03-22
13. Tsoenabler for linux (2023), https://github.com/cyyself/m1tso-linux – accessed 2023-03-26
14. Ali, Z., Tanveer, T., Aziz, S., Usman, M., Azam, A.: Reassessing the performance of arm vs x86 with recent technological shift of apple. In: 2022 International Conference on IT and Industrial Technologies (ICIT). pp. 01–06 (2022). https://doi.org/10.1109/ICIT56493.2022.9988933
15. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: Seidl, H. (ed.) ESOP. pp. 26–46. Lecture Notes in Computer Science, Springer-Verlag (2021)
16. Boehm, H.J., Adve, S.V.: Foundations of the c++ concurrency memory model. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 68–78. PLDI '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375591, https://doi.org/10.1145/1375581.1375591
17. Dubois, M., Scheurich, C., Briggs, F.: Memory access buffering in multiprocessors. In: Proceedings of the 13th Annual International Symposium on Computer Architecture. p. 434–442. ISCA '86, IEEE Computer Society Press, Washington, DC, USA (1986)

18. Flur, S., Sarkar, S., Pulte, C., Nienhuis, K., Maranget, L., Gray, K.E., Sezgin, A., Batty, M., Sewell, P.: Mixed-size concurrency: Arm, power, c/c++11, and sc. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 429–442. POPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009839, https://doi.org/10.1145/3009837.3009839
19. Gharachorloo, K., Gupta, A., Hennessy, J.: 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. p. 245–257. ASPLOS IV, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/106972.106997, https://doi.org/10.1145/106972.106997
20. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. SIGARCH Comput. Archit. News **18**(2SI), 15–26 (may 1990). https://doi.org/10.1145/325096.325102, https://doi.org/10.1145/325096.325102
21. Goodman, J.R.: Cache consistency and sequential consistency (1991), http://digital.library.wisc.edu/1793/59442 – accessed 2023-03-28
22. Gupta, N., Ashiwal, R., Brank, B., Peddoju, S.K., Pleiter, D.: Performance evaluation of parallex execution model on arm-based platforms. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER). pp. 567–575 (2020). https://doi.org/10.1109/CLUSTER49012.2020.00080
23. Higham, L., Kawash, J., Verwaal, N.: Defining and comparing memory consistency models (1997)
24. Johnson, D.: Apple M1 Microarchitecture Research (2023), https://dougallj.github.io/applecpu/firestorm.html – accessed 2023-03-23
25. Kenyon, C., Capano, C.: Apple silicon performance in scientific computing. In: 2022 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–10 (2022). https://doi.org/10.1109/HPEC55821.2022.9926315
26. Kodama, Y., Kondo, M., Sato, M.: Evaluation of spec cpu and spec omp on the a64fx. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER). pp. 553–561 (2021). https://doi.org/10.1109/Cluster48925.2021.00088
27. Lamport: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **C-28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
28. Mattioli, M.: Meet the fam1ly. IEEE Micro **42**(3), 78–84 (2022). https://doi.org/10.1109/MM.2022.3169245
29. Naeem, A., Chen, X., Lu, Z., Jantsch, A.: 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). pp. 154–159 (2011). https://doi.org/10.1109/ASPDAC.2011.5722176
30. Ouro, P., Lopez-Novoa, U., Guest, M.F.: On the performance of a highly-scalable computational fluid dynamics code on amd, arm and intel processor-based hpc systems. Computer Physics Communications **269**, 108105 (2021). https://doi.org/https://doi.org/10.1016/j.cpc.2021.108105, https://www.sciencedirect.com/science/article/pii/S0010465521002174
31. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. Proc. ACM Program. Lang. **2**(POPL) (dec 2017). https://doi.org/10.1145/3158107, https://doi.org/10.1145/3158107

32. SPARC International, Inc., C.: The SPARC Architecture Manual: Version 8. Prentice-Hall, Inc., USA (1992)
33. SPARC International, Inc., C.: The SPARC Architecture Manual (Version 9). Prentice-Hall, Inc., USA (1994)
34. Xia, J., Cheng, C., Zhou, X., Hu, Y., Chun, P.: Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services. IEEE Micro **41**(5), 67–75 (2021). https://doi.org/10.1109/MM.2021.3085578