Chapter 9
Test Generation and Lightweight Checking for Multi-core Memory Consistency

Doowon Lee and Valeria Bertacco

9.1 Introduction

Modern microprocessor chips integrate multiple cores to provide high computing capability under a limited power budget. Multi-core processors are indeed widely deployed in consumer electronics and data centers, and the number of cores integrated in a single chip is ever growing since the inception of multi-core architecture. For example, in the server domain, a recent Intel Xeon Skylake-SP processor integrates 28 powerful cores, each of which supports 2 threads of execution. In the mobile system-on-chip domain, a recent Qualcomm's Snapdragon consists of 8 single-threaded cores, as well as a variety of heterogeneous application-specific accelerators. For these multi-core processors, the software can be designed to utilize the multiple cores available in two different ways: either multiple independent tasks run separately on each core, or a single task (application) spawns multiple threads of execution, each assigned to a different core. In this latter case, the threads must share the same memory space (multithreading). In recent years, multithreading has become increasingly popular, as it collaboratively advances a single application with the potential of greatly reducing the application's completion time.

Multi-core systems deploy a complex memory subsystem to enable efficient multiprocessing and multithreading computation, by processing tens or hundreds of memory operations in parallel. The memory subsystem typically includes multiple

This work is based on an earlier work: "MTraceCheck: Validating Non-Deterministic Behavior of Memory Consistency Models in Post-Silicon Validation" by Doowon Lee and Valeria Bertacco in the 44th Annual International Symposium on Computer Architecture (ISCA '17) ©ACM 2017. https://doi.org/10.1145/3079856.3080235.

D. Lee · V. Bertacco (✉)
University of Michigan, 2260 Hayward Street, Ann Arbor, MI 48109-2121, USA
e-mail: valeria@umich.edu

D. Lee
e-mail: doowon@umich.edu

P. Mishra and F. Farahmandi (eds.), *Post-Silicon Validation and Debug*,
https://doi.org/10.1007/978-3-319-98116-1_9

caches and memory channels. These memory components interact with the processor cores via on-chip communication networks such as buses, crossbars, and mesh networks. In this complex memory subsystem architecture, many memory-access requests can be executed in an arbitrary order that is different from the sequential order defined in the program code. This memory-level parallelism, on one hand, is an essential element of a high-performing multi-core system. On the other hand, it comes with great verification and validation challenges. There are various memory-access interleaving patterns that the system can exhibit—many of which are difficult to foresee during design time.

Various microprocessor vendors define their own specifications on correct memory ordering behaviors, called memory consistency models. There are a wide range of memory consistency models, ranging from weak models, as used by ARM and IBM, to stronger ones, like the one adopted by Intel, to the strongest model of MIPS systems. A weak memory consistency model allows memory operation re-orderings that are disallowed by a stronger memory model. From the viewpoint of hardware verification, every memory ordering pattern that an implementation may exhibit must be checked to determine whether it abides by the memory consistency model. Note that, however, modern processor designs are usually too complex to be formally verified within the relatively short amount of time allowed for verification and validation. As a result, functional bugs related to memory consistency have slipped through mainstream multi-core processors, as reported in many specification update documents [8, 28].

9.1.1 Brief Introduction to Memory Consistency Models

Memory consistency models (MCMs) are the specifications regarding memory ordering behaviors in multi-core systems, providing a protocol between hardware designers and software programmers [2, 46]. A multi-core microprocessor can arbitrarily re-order its memory operations under execution, as long as such re-ordering obeys the memory ordering rules specified by the MCM. At the same time, programmers must understand the microprocessor's re-ordering behaviors to write correct multithreaded programs.

In addition, every execution of a multithreaded program can exhibit different memory-access interleaving patterns, depending on micro-architecture state at the time the program is executed. In order to tame such nondeterministic memory interleavings, programmers must insert fence operations whenever necessary to ensure correct program execution. A fence operation is a special operation that restricts memory re-ordering behaviors—the order among memory operations before and after a fence operation is strictly enforced. In other words, memory operations issued before the fence operation must be completed before those following the fence operation can be issued.

There are several memory consistency models that are worth noting. Here, we briefly introduce only three groups of models; please refer to [2, 46] for a more

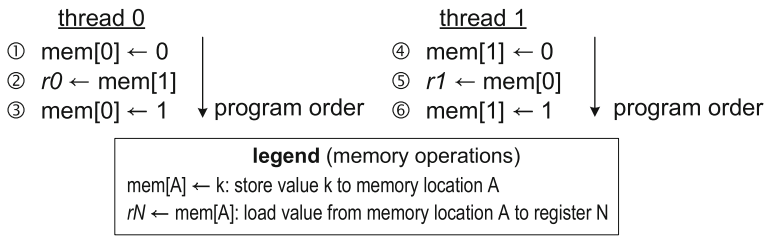

Fig. 9.1 A 2-threaded test sharing 2 memory locations. Under the total store order memory model, load operations ② and ⑤ are allowed to perform in advance of preceding store operations ① and ④, respectively

in-depth discussion. *Sequential consistency* (SC) [31] is the most intuitive, strongest memory model among most models in the literature. No memory re-ordering within a thread is allowed by this model, so memory operations are performed in the order specified by the program. However, this model still allows an arbitrary interleaving of memory operations across different threads—the inter-thread interleaving is the source of nondeterministic results of memory operations in this model.

Total store order (TSO) [27, 50] allows load operations to be performed before preceding store operations, relaxing the store→load ordering requirement. Figure 9.1 illustrates a two-threaded program sharing two memory locations. In each thread of this program, the load operation (second operation) can be performed before its preceding store operation (first operation). This re-ordering allows the load operation to retrieve data from caches or main memory components while the store operation still resides in the core. Relaxing the store→load ordering enables some micro-architectural optimizations, such as store buffering, to greatly enhance memory-level parallelism. Note that, however, the last store operation (third operation) cannot be performed before any of its preceding operations, meaning that the store→store and load→store orderings are still preserved in this model. Many multi-core processors, such as Oracle SPARC and Intel Core and Xeon processors, deploy the TSO memory model or a variation of the same.

Weakly ordered memory models [10, 11, 25, 49] are also widely adopted by many processors such as ARM Cortex and IBM POWER processors. In this group of memory models, any memory operation can be performed in advance of its preceding memory operation in the absence of a fence. Multi-core processors with this kind of memory models can deploy a variety of micro-architectural optimizations to fully utilize the memory bandwidth and reduce the memory-access latency without being subject to memory ordering rules.

There are slight differences among memory models within the same MCM category. For example, Intel processors allow the values written by store operations to be observed as occurring in different orders by different threads [27], while SPARC processors do not allow such different orderings [50]. For weakly ordered memory models, the ARMv8 architecture provides many variants of the memory barrier operation (`dmb`) for different operation types (load and store) and memory domains

(non-shareable, inner shareable, outer shareable, and full system) [11]. IBM POWER v2.07 supports a few different types of fence operations (`sync`, `lwsync`, `ptesync`, `eieio` and `mbar`), each serving a different memory ordering function [25].

9.1.2 Memory Consistency Verification and Validation

Ensuring that the memory-access patterns observed by the memory consistency model stipulated for the architecture under consideration can be achieved by various verification and validation efforts. In the early design stages, the memory consistency model of a multi-core processor is determined by computer architects. They decide how the processor's micro-architecture should enforce memory ordering rules to follow the memory consistency model. Therefore, the goal of memory consistency verification and validation is to check that the processor implementations (e.g., functional simulators, RTL models, and silicon chips) respect the memory ordering rules.

Formal Verification of Memory Consistency

Formal methods have been applied to detect memory consistency violations in the literature. With formal methods, the memory subsystem under verification is modeled as mathematical formulas that capture memory ordering behaviors of the memory subsystem. For example, Alglave [3, 6] uses the Coq theorem prover [47] to verify several weak MCMs, showing that it can expose a subtle memory consistency bug in the ARM Cortex-A9 processor [8]. Expanding this Coq-based framework, PipeCheck [34] specifies micro-architectural behaviors in detailed happens-before graphs that capture interactions among pipeline stages. RTLCheck [39] automatically generates SystemVerilog Assertions from micro-architectural axioms and performs formal property verification using JasperGold.

Litmus tests [4, 5, 7, 36, 37] are often paired with the formal methods mentioned earlier. These tests are a small set of multithreaded programs that are tailored to trigger specific memory consistency bugs when they are present. A typical litmus test includes several instructions with a few threads of execution, and targets a very specific consistency violation scenario. For a given litmus test and mathematical model of a processor design, all possible memory ordering patterns are explored and checked for the targeted violation scenario. Figure 9.2 illustrates a memory consistency violation that can be detected by a 2-threaded litmus test, as the one shown on the left. While litmus tests are very powerful in uncovering many consistency violations, they are often insufficient to validate full multi-core systems [20]. This is partly because litmus tests often miss very subtle corner cases that have not been captured by the mathematical model. In other words, an inaccurate model often leads to false positives (i.e., falsely proven to be correct) [44].

Dynamic Validation of Memory Consistency

Dynamic validation methods check memory ordering patterns observed in processor implementations while *running* multithreaded programs. This validation approach

uses various types of test programs, such as litmus tests, constrained-random tests, and real applications. The major difference with formal verification is that dynamic validation analyzes the results of *concrete execution* in *real design implementations*. In other words, dynamic validation does not involve mathematical models, exhaustive enumeration, nor symbolic execution.

Hangal et al. [24] present an industrial memory consistency validation tool, called TSOtool, that is used to validate SPARC chip-multiprocessor systems [50]. Roy et al. [43] try to speed up memory consistency validation by approximating the result-checking algorithm. Meixner and Sorin [40] include small additional hardware modules within a memory subsystem so as to track memory ordering behaviors observed at runtime. Similar hardware modifications on the memory subsystem have been proposed by other research works [16, 38]. In addition, Axe [41] provides a memory trace generator that can be applied to the Berkeley's RISC-V and Cambridge's BERI memory subsystems. It also provides an open-source memory consistency checker supporting various memory models.

Constraint Graph

In order to check the correctness of memory ordering, the order among memory operations is analyzed using their happens-before relationships. Each happens-before relationship indicates a partial order of two memory operations. By collecting all happens-before relationships, we construct a constraint graph, also known as a happens-before graph, where each vertex corresponds to a memory operation, and each edge represents the happens-before relationship between two vertices. If there is a cycle in a constraint graph, it implies a memory consistency violation. How to construct a constraint graph has been extensively investigated in the literature [3, 15–17, 34, 43].

In Fig. 9.2, the constraint graph on the right represents the result of the 2-threaded litmus test on the left. This constraint graph is built under the assumption that the system adopts a TSO memory model. As the graph shows, there is a cyclic dependency along the four memory operations, thus the execution violates the TSO model. In general, to check for cyclic dependencies in a constraint graph, we can use a topological sorting or depth-first search, which has high computational complexity, $\Theta\,(V + E)$, where V and E are the set of vertices and edges, respectively [18]. Therefore, the

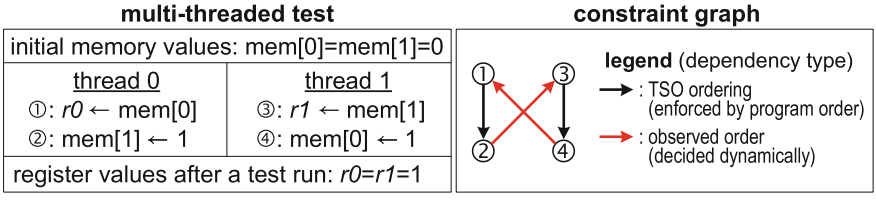

Fig. 9.2 A 2-threaded test execution (left) and its corresponding constraint graph (right). The TSO model allows various memory operation results, except for the $r0{=}r1{=}1$ shown on the bottom left. The constraint graph on the right identifies this violation with a cyclic dependency

result-checking can quickly become the critical bottleneck of the entire memory consistency validation process.

9.1.3 Post-silicon Microprocessor Validation

Post-silicon microprocessor validation strives to detect subtle errors that slip through silicon chips. In this validation stage, constrained-random tests are widely used along with directed tests. Constrained-random tests are essential to verify unexpected use cases that have not been identified by verification and validation engineers, while directed tests target corner cases that are expected by the engineers. These two types of tests are often designed in a way that overcomes the challenge of limited observability in post-silicon validation. Specifically, these tests can be created in a way such that their test results can be easily checked on the same platform that is under validation (self-checking). Self-checking solutions [1, 21, 33, 48] need to observe only minimal test results to check the correctness of test runs, and thus they mitigate the overhead of transferring test results to a host machine. For example, QED [33] duplicates instructions and compares the results of the original and the duplicate instructions. Reversi [48] generates a sequence of reversible instructions where the result of the sequence becomes a simple value to be checked easily—for instance, it uses a pair of addition and subtraction operations with the addend and the subtrahend (i.e., the number to be subtracted) being equal. Foutris et al. [21] leverage the fact that an instruction set consists of various instructions, most of which can be replaced by a sequence of other instructions—for instance, a series of additions can replace a multiplication instruction.

Memory Consistency Post-silicon Validation

Both directed tests and constrained-random tests are also commonly used in memory consistency validation. Litmus tests are popular directed tests for memory consistency validation. These are often hand-crafted by engineers [7, 26] or can be automatically generated from a formal model of a memory subsystem [5, 36]. Constrained-random tests are tailored to exhibit rare memory ordering behaviors, by intensively accessing a small number of memory locations shared across multiple threads. These tests are often generated in a way that every store operation writes a unique value into memory so that store operations can be easily disambiguated. Also, in these tests, a partial order of a load operation is determined based on the value read by the load operation, which is captured by a *reads-from* relationship. A reads-from relationship indicates a direct happens-before relationship between some load and store operations: "if a load operation reads from a store operation, the store must happen before the load." With reference to Fig. 9.2, the load operation ③ reads from the store operation ②, which implies that ② happens before ③.

Both litmus tests and constrained-random tests are run repeatedly to strive to exhibit various distinct memory-access interleaving behaviors. There are many different valid memory-access interleaving outcomes, especially for constrained-random

tests where many more memory operations are included. In other words, every run of a test may lead to a unique memory-access interleaving behavior that has not been seen by prior test runs. Consequently, the activity of validating the test outcomes requires much more analysis and computation than other post-silicon validation activities, where test outcomes are deterministic and can be self-checked. Thus, it is important to minimize the result-checking computation to improve the overall efficiency of the validation process.

In the rest of this book chapter, we present MTraceCheck, an efficient memory consistency validation framework specialized for post-silicon validation [32].

9.2 MTraceCheck: Efficient Memory Consistency Validation

MTraceCheck is a memory consistency validation framework that efficiently checks many distinct memory ordering patterns. It is tailored to post-silicon validation environments with very limited observability—however, it is also applicable to pre-silicon dynamic validation environments such as instruction set simulation, RTL design simulation, or FPGA prototyping.

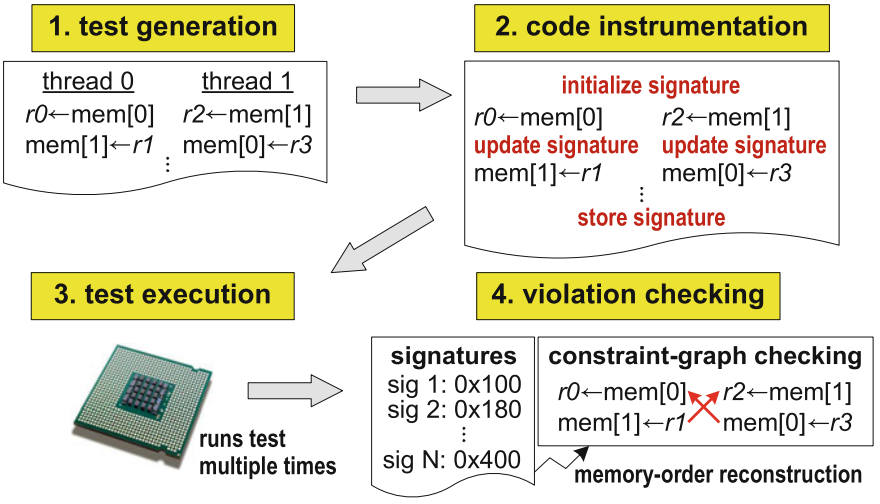

Fig. 9.3 MTraceCheck overview. MTraceCheck first generates multithreaded constrained-random tests, which are then instrumented with our observability-enhancing code. The instrumented code computes a memory-access interleaving signature at runtime. Each instrumented test is run multiple times to obtain a collection of signature values. These signature values are then decoded to reconstruct constraint graphs capturing happens-before relationships among the memory accesses. The constraint graphs are then collectively validated by our checking algorithm that leverages the similarities among the graphs

Figure 9.3 outlines the validation workflow using the MTraceCheck framework in four steps. In the first step, it generates constrained-random tests using several test-generation parameters. The generated tests include multiple threads of execution where each thread performs many memory operations on a small memory region shared across all threads, as in [20, 24]. In the next step, the generated tests are augmented with signature computation code that calculates a signature value corresponding to the memory-access interleaving behaviors observed at runtime. We explain this code instrumentation process in Sect. 9.2.1. The augmented tests are then loaded onto the platform under validation and repeatedly executed on the platform. From the repeated test runs, we collect signatures and store them in a thread-local storage. Lastly, we examine the signatures in a collective manner that minimizes repetitive checking computation. This checking process is detailed in Sect. 9.2.2. Compared to prior related work on post-silicon memory consistency validation, MTraceCheck provides the following contributions:

- MTraceCheck's *code instrumentation process* computes a signature representing the order among memory operations. The signature minimizes the amount of data transfers for the purpose of result-checking.
- MTraceCheck's *lightweight graph-checking algorithm* leverages the similarities of constraint graphs across repeated executions of a memory consistency test.

In Sect. 9.3, we quantify the benefits and drawbacks of MTraceCheck in two bare-metal systems: an x86 quad-core desktop and an ARM octa-core single-board computer. We also demonstrate that MTraceCheck is capable of finding subtle memory consistency bugs in the gem5 full-system simulator in Sect. 9.4.

9.2.1 Enhancing Observability by Code Instrumentation

As we discussed in Sect. 9.1.3, memory consistency tests result in a variety of memory ordering patterns. To capture the memory ordering patterns, we need to trace the value read by each load operation. The collection of loaded values then can be used to reconstruct happens-before relationships among the memory operations performed in the test. During the validation process, the loaded values can be traced by using hardware debug features in place. For example, ARM's Embedded Trace Macrocell (ETM) [9] can record the instructions and data at runtime. Similarly, Intel provides the Branch Trace Store (BTS) feature [27] that can track the history of branches. However, memory consistency validation requires a huge amount of memory operations to be traced on the fly, and using these features is often insufficient to do so. Specifically, ETM's on-chip trace buffers are usually small—tests must be halted whenever the trace buffers become full. Also, BTS shares the memory subsystem to store the results of branches, thus potentially perturbing the memory ordering behaviors of test execution. Therefore, in this work, we strive to trace the memory operations efficiently for the purpose of memory consistency validation. In addition,

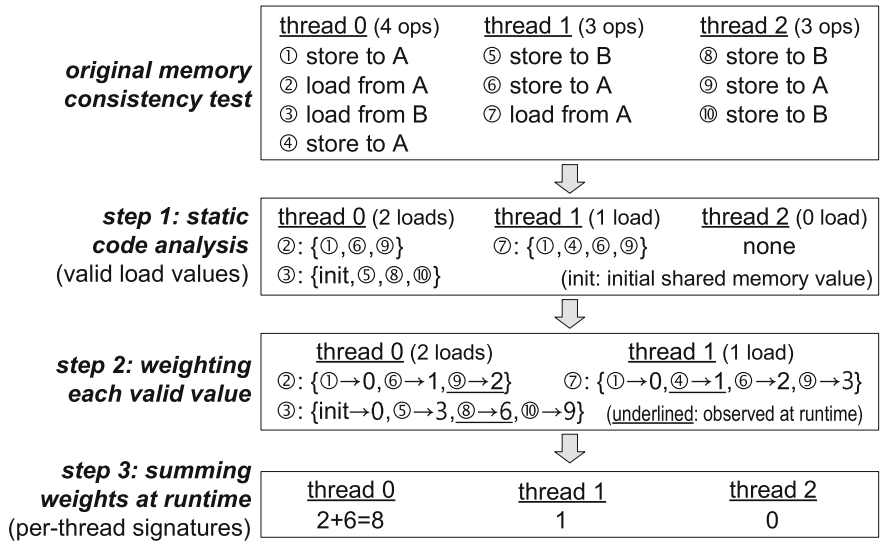

Fig. 9.4 Memory-access interleaving signature. Each signature value represents a unique memory ordering pattern observed during the test run. To compute the signature, we first profile all values that can be read by each load operation. We then assign an integer weight for each of the values. These two steps are performed statically. At runtime, the weights of the observed values are accumulated for each thread independently, creating a per-thread signature. Finally, all per-thread signatures are concatenated to create an execution signature

we advocate a software-based technique in tracing the memory operations, which can be also paired with these existing hardware debug features.

A conventional software-based memory-tracing method has been shown in TSOtool [24]. In TSOtool, the values read by load operations are temporarily stored in registers, and then flushed back to memory. Note that, due to the limited amount of architectural registers available, this register-flushing method frequently interrupts the test execution. In addition, it is bound to perform an excessive amount of store operations to record the history of loaded values. These store operations are subject to perturb the original test's memory-access behaviors, altering the intention of the original test. In summary, we need to minimize the amount of store operations required to trace the results of the original test's memory operations.

9.2.1.1 Computing Memory-Access Interleaving Signature

Here, we describe our signature computation method to encapsulate the results of memory operations in a small signature value. Our method is inspired by a prior control-flow profiling method proposed by Ball and Larus [13]. This method presents a systematic way to compute a small sum of values accumulated for each basic block executed. We take a similar approach to compute a signature representing the results of memory operations.

As mentioned earlier, the values read by load operations need to be traced to decide the order among memory operations, or memory-access interleaving behaviors. In this work, we propose a *memory-access interleaving signature* that expresses the loaded values in a compact format. Each distinct value in the memory-access interleaving signature indicates a unique memory ordering pattern of a multithreaded program. This small-sized signature is stored for each test run; unlike the register-flushing technique [24], our signature-based solution introduces only a handful of store operations for the purpose of validation. By doing so, we aim at minimizing the interference of memory tracing in order to observe the results of memory operations while running the test.

To this end, we compute a signature as we run a memory consistency test. Figure 9.4 summarizes our signature computation process in three steps. In the first step, we analyze the original test statically so as to collect all possible values that can be read by each load operation. The top box of the figure illustrates a 3-threaded test program created by a constrained-random test generator. The test program includes 10 memory operations, which share 2 memory addresses (`A` and `B`). Among the 10 operations, there are 3 load operations (②, ③, and ⑦), each of which can take any of the values listed in the set in the second box of the figure.

Analyzing all possible loaded values can be a demanding task depending on the test under validation. For constrained-random tests, test generators usually have all the information on the generated operations including the address of each memory operation. Thus, memory addresses can be extracted from the test generators and our static analysis can be easily done. However, in real-world multithreaded programs (e.g., graph analytics), our static analysis can be limited and some memory addresses cannot be disambiguated. In this case, we can use a binary instrumentation tool in order to disambiguate memory addresses as much as possible via profiling. We can also exclude memory operations with unknown addresses from the signature computation, and record the results of these operations as the conventional register-flushing method does.

Algorithm 1 Weight assignment

Input: all possible $reads\_from$ relationships
Output: $weights, multipliers, reads\_from\_maps$
$multiplier \leftarrow 1$
for each load operation L from first to last in test program **do**
  $multipliers[L] \leftarrow multiplier$
  $index\_reads\_from \leftarrow 0$
  for each $reads\_from$ for the load operation L **do**
    $weights[L][index\_reads\_from] \leftarrow index\_reads\_from \times multiplier$
    $reads\_from\_maps[L][index\_reads\_from] \leftarrow reads\_from$
    $index\_reads\_from \leftarrow index\_reads\_from + 1$
  end for
  $multiplier \leftarrow multiplier \times index\_reads\_from$
end for
return $weights, multipliers, reads\_from\_maps$

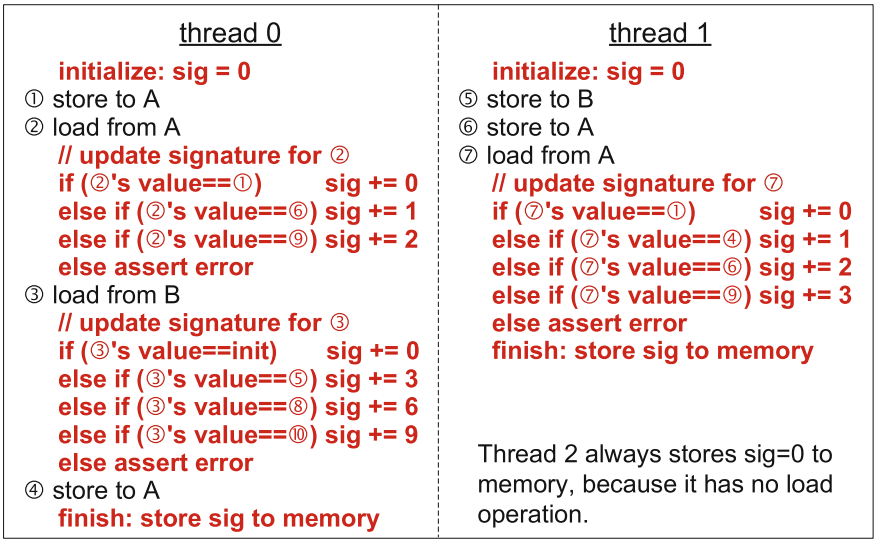

Fig. 9.5 Code instrumentation. Branch and arithmetic instructions are instrumented after each load operation, and these instructions update the signature variable (`sig`) as the test runs. When the test finishes, the computed signature is stored in a thread-local memory region

In step 2, we decide an integer weight for each possible value read by load operations. This step is illustrated in the third box of the figure and also summarized in Algorithm 1. We perform the weight assignment for each thread individually, while the previous step is performed globally considering all threads. With reference to thread 0 in the figure, the first load operation ② has three possible loaded values (or three possible reads-from relationships), stored by ①, ⑥, and ⑨. We assign weights 0, 1, and 2 for these values, respectively. For the next load operation ③, we assign multiples of 3 since there were three options in the previous load operation. The way we allocate the weights guarantees a unique correspondence between signatures and the orders of memory operations. Our weight assignment resembles the way that [13] assigns an integer value for each edge in a control-flow graph.

In the last step, we accumulate the weights that correspond to the loaded values at runtime. Suppose in the third box in the figure, the value written by ⑨ (weight 2) is read by the first load operation and the value written by ⑧ (weight 6) is read by the second load operation, as illustrated by the two underlined values in thread 0 of the box. As a signature for this thread, we obtain the sum of the two weights, which is 8, at the end of the execution. Similarly, suppose that the load operation ⑦ in thread 1 reads the value underlined, and its weight (1) becomes a signature of this thread. Thread 2 has no load operation, so it always returns 0 as its signature. These per-thread signatures are illustrated in the bottom box of the figure. Note that the signature computation is done for each thread individually at runtime. At the end of the test run, we form the *execution signature* by concatenating all the per-thread signatures obtained for each test run.

Code Instrumentation

Figure 9.5 shows the test program after we conduct our code instrumentation to the original test in Fig. 9.4. The instrumented code includes three parts. First, the `sig` variable is initialized to 0 at the beginning of the test. The `sig` variable is then increased after each load operation. This increase is carried out by the second instrumented section: a chain of branch and addition operations conditioned on the loaded value. Note that we append an assertion statement at the tail of each branch chain. This assertion statement is intended to catch obvious errors that need no constraint-graph checking. For instance, it can catch a program-order violation that the load operation ② takes the value from the younger store operation ④. In the last segment of instrumented code, the accumulated signature `sig` is stored in a thread-local storage when the test finishes.

Depending on the memory-interleaving possibilities, the signature can exceed the size of a single register (typically 32 bits or 64 bits), in which case we save the `sig` variable before it overflows, reinitialize it (i.e., `sig`=0), and start accumulating it from there. In other words, we split a per-thread signature into multiple words. To this end, we statically identify any signature overflow when we compute weights during our code instrumentation process. In this case, we store the current signature (i.e., the signature that has been computed until the point we detect the overflow) to a thread-local memory region, then start over the signature computation by resetting the *multiplier* (i.e., *multiplier*=1). By doing so, the previous signature represents the previous code segment, while the new signature represents the next code segment.

While our instrumentation increases the code size to some degree, its impact on test runtime is much smaller compared to the increase of the code size. We provide experimental evaluations on the code size and runtime overheads later in Sect. 9.3. Note that the instrumented routines after load operations minimally perturb the memory-access patterns intended by the original test, because the routines are composed of mostly branch and arithmetic operations, but not memory operations. There are two exceptional cases where memory-access patterns can be affected: (1) when the signature `sig` needs to be reset due to an overflow possibility as mentioned earlier—in this case, the previous `sig` is flushed to memory before it is cleared, and (2) when the program execution hits any of the assertion statements (`assert error`). Note that modern microprocessors are equipped with out-of-order execution engines with highly accurate branch predictors, so our instrumented routines can be effectively executed in parallel with memory operations.

9.2.1.2 Reconstructing the Order Among Memory Operations

We decode each signature following the pseudocode described in Algorithm 2 so as to reconstruct the order among memory operations captured by a signature. The decoding process is basically a reverse process of the signature computation process in Algorithm 1. The goal of the decoding process is to obtain a set of reads-from relationships represented by a signature. Before we apply the decoding procedure in Algorithm 2, we first split an execution signature into a set of per-thread signatures. For each thread, we then reconstruct the reads-from relationships, one at a time,

Algorithm 2 Signature decoding procedure

```
1: Input: signature, multipliers, reads_from_maps
2: Output: reads_from relationships observed at runtime
3: for each load L from last to first in test program do
4:     multiplier ← multipliers[L]
5:     index_reads_from ← signature / multiplier
6:     signature ← signature % multiplier
7:     reads_from[L] ← reads_from_maps[L][index_reads_from]
8: end for
9: return reads_from
```

starting from the last load operation in the thread. The reconstruction process walks backward to the first load operation in the thread (line 3 in the algorithm). The *multipliers* array has the multiplier used to calculate each load operation's weights. For each load operation, the *signature* is divided by the multiplier specified in the *multipliers* array (lines 4 and 5). The quotient indicates the index of a reads-from relationship observed for the load operation (line 5). The remainder represents a signature value before being accumulated for this load operation, and thus it replaces the *signature* for the next load operation under reconstruction (line 6). The index of the reads-from relationship is then used to look up the *reads_from_maps* array which provides the mapping relationships between indices and reads-from relationships (line 7).

At the same time we assign weights, we gather additional information necessary to reconstruct the constraint graph. Besides the *multipliers* and *reads_from_maps* arrays, we also collect the intra-thread memory ordering required by the MCM (e.g., ①→② shown in Fig. 9.2), and the write-serialization order (e.g., store operations to the same address within a thread must be observed in the program order, such as ① and ④ in Fig. 9.4). All the aforementioned information is necessary to reconstruct constraint graphs as well as reads-from relationships.

9.2.1.3 Signature Size

The size of a signature is proportional to the number of reads-from relationships across all memory operations. Specifically, our memory-access interleaving signature is designed to capture all allowed memory-access interleaving patterns under a very weak MCM where no memory ordering is enforced. In a stronger MCM where fewer memory-access interleavings are allowed, however, our signature can express a value corresponding to a memory ordering *disallowed* by the MCM, making the signature unnecessarily large for the MCM. These invalid signature values will be eventually caught as consistency violations during our graph checking in Sect. 9.2.2. In addition, we try to keep our signature small by considering intra-thread program-order requirements. For example, by a program-order requirement, a load operation cannot take a stale value from a store operation older than the most recent store

operation within the same thread. This kind of violations is caught by the assertion statements in Fig. 9.5.

Here we estimate the signature size of a test generated under a constrained-random testing environment used in our experimental evaluations. In our testing environment, we have several test-generation parameters specifying the characteristics of a test: the number of threads (T), the number of stores per thread (S), the number of loads per thread (L), and the number of shared memory locations (A). We created memory addresses in a random fashion. With these parameters, the size of a per-thread signature can be expressed by the following equation:

$$\texttt{per-thread signature cardinality} = \left\{1 + \frac{S}{A}(T-1)\right\}^{L} \quad (9.1)$$

The curly brackets in the equation include two components for each load operation. The first component (1 in the equation) captures the case where the load reads from the same thread (i.e., reading the value written by the most recent store to the same address in the same thread), or the case where the load reads the initial memory value if there is no prior store operation to the same address. The second component ($\frac{S}{A}(T-1)$) represents the cases where the load reads from any of the other threads. $\frac{S}{A}$ indicates the number of store operations with the same address as the load. The expression enclosed by the curly brackets is then multiplied L times to compute all possible combinations across all load operations in the thread. The size of an execution signature is T times the equation above, because an execution signature is the collection of all per-thread signatures.

As an example, the smallest test configuration we used (Table 9.2) has the following test-generation parameters: $T = 2$, $S = L = 25$, $A = 32$. A per-thread signature in this configuration can represent $\{1 + \frac{25}{32}(2-1)\}^{25} \approx 1.9 \times 10^6 \approx 2^{21}$ sets of loaded values, requiring approximately a 21-bit storage. In Sect. 9.3.4, we will show the average size of the execution signature across various constrained-random tests, and discuss how to reduce the signature size in Sect. 9.5.

9.2.2 *Collective Graph Checking*

To check the correctness of memory ordering observed at runtime, we analyze the happens-before relationships among all memory operations performed during test execution. The happens-before relationships, as studied in [3, 34], can be classified into three types of observed happens-before relationships as follows: *reads-from*, *from-read*, and *write-serialization*. Besides these observed relationships, we take into account intra-thread ordering rules as defined by the MCM. With reference to Fig. 9.6, which assumes the TSO memory model, the store operation ① should happen before ③ as illustrated with the *TSO ordering* edge. But ① does not necessarily happen before ②, so there is no edge between these two operations. Also, suppose that the load operation ② takes the value of the store operation ❶; this happens-before

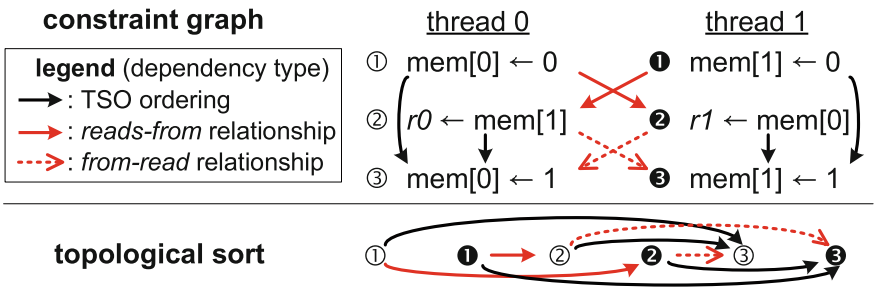

Fig. 9.6 A constraint graph (top) and its topological sort (bottom). No consistency violation can be found from this constraint graph because there exists a topological sort for the graph, following all happens-before constraints

relationship is captured by the red reads-from edge between the two operations in the figure. From this reads-from relationship, we can derive that a *from-read* relationship to the next store operation ❸ in thread 1, because store operations to the same address must be ordered (*write-serialization*).

Once we construct a constraint graph with happens-before relationships, we check for a cyclic dependency along the relationships. For instance, suppose three happens-before relationships among three memory operations (A, B, and C) in the following: (1) A happens before B, (2) B happens before C, and (3) C happens before A. These three relationships can be summarized as A→B→C→A, which is a cyclic dependency. This cyclic dependency is an indication of memory consistency violation.

Topological sorting, as introduced in Sect. 9.1.2, is a conventional method to detect a cycle in a graph, by trying to find an order among all the vertices in a way that there is no backward edge. In a formal definition, *a topological sort of a directed graph is a linear ordering of all its vertices such that if the graph contains an edge* (u, v), *then* u *appears before* v *in the ordering* [18]. The constraint graph in Fig. 9.6 can be topologically sorted as shown on the bottom of the figure, so we find no memory consistency violation in this example. However, the constraint graph in Fig. 9.2 has no topological sort, indicating a consistency violation.

Prior works apply topological sorting to each individual constraint graph obtained from post-silicon memory consistency tests. However, we notice that *many test runs actually exhibit memory-access interleaving patterns that are similar to each other*. This observation leads us to our novel *collective* constraint-graph checking solution that exploits the similarities among test runs. With our collecting checking solution, we strive to reduce the amount of computation required to validate the results of memory consistency tests. In the following subsections, we discuss how to estimate the similarities among constraint graphs (Sect. 9.2.2.1), then present our topological re-sorting technique (Sect. 9.2.2.2).

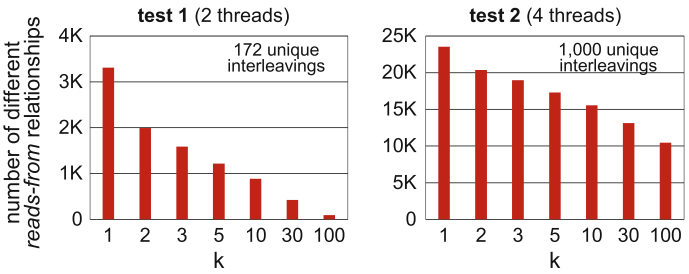

Fig. 9.7 Measuring similarities among constraint graphs using k-medoids clustering for two tests (one with 2 threads and the other with 4 threads). The medoid graphs are a set of "representative" constraint graphs. In order for the medoids to be truly representative, the number of differing reads-from relationships should be very small. We found that as k increases, the number of differing reads-from relationships decreases as shown in both tests. However, many reads-from relationships in test 2 remain different from the medoid graphs, even in a large k value. In other words, the cluster tightness is very low because of many memory-access interleaving opportunities in this test

9.2.2.1 Estimating the Similarities Among Constraint Graphs

As mentioned earlier, we would like to reduce the computation required to detect cyclic dependencies in constraint graphs. To this end, we propose an incremental verification scheme—we first compare a graph under validation with another graph that was validated previously. We then isolate the portion of the new graph that differs from the previous graph, and check only the differing portion by applying our modified topological sorting.

Limit Study—k-Medoids Clustering to Evaluate Graph Similarities

In the process of designing our incremental checking scheme, we first evaluated the similarities among graphs obtained from the same test. To this end, we conducted a preliminary study to identify a small set of representative graphs from the entire set of constraint graphs generated from repeated executions of two constrained-random tests. The first test includes 2 threads, 50 memory operations per thread, and 32 distinct shared memory addresses. The second test has more threads (4 threads), but the rest of the parameters are identical to the first test. For each of the two tests, we applied a k-medoids clustering analysis [29] to the set of constraint graphs obtained from 1,000 test runs so as to choose k representative graphs. In this analysis, our distance metric was the number of different reads-from relationships between the two graphs. To obtain reads-from relationships in this preliminary experiment, we used an in-house architectural simulator that mimics the behavior of sequential consistency (SC) [31]. This simulator randomly chooses memory operations without violating SC.

Figure 9.7 shows the total number of different reads-from relationships, where each graph is compared to its closest medoid graph. In the first test, where only 172 executions out of 1,000 executions are unique, the number decreases rapidly as k increases. On the contrary, in the second test, where every execution results in a

unique interleaving behavior, the number still remains high even in the highest k value ($k = 100$). In other words, the selected medoid graphs are still vastly different from the individual graphs.

From these experiments, we conclude that the k-medoids clustering is unfit for two reasons in the following. (1) There are a large amount of discrepant reads-from relationships after applying this clustering algorithm. (2) The computational complexity of k-medoids clustering is very high [29], exceeding the complexity of topological sorting. Therefore, using this clustering negates the potential time saving of a fast topological sorting that exploits graph similarities.

Our Solution—Sorting Signatures and Diffing Corresponding Graphs

Instead of the k-medoids analysis, we strive to leverage a lightweight computation to find a previous graph sufficiently similar to a graph under validation. To this end, we repurpose our memory-access interleaving signature (Sect. 9.2.1) as a graph similarity metric.

To be specific, we first collect all execution signatures from multiple executions of a test. We then *sort* these execution signatures in ascending order. Note that signatures adjacent in this order have reads-from relationships similar to each other. The difference between two adjacent signatures is likely to appear on low bits of the signatures, which translate to differing reads-from relationships at the beginning of the test. We then reconstruct a constraint graph for each of the signatures in the sorted order. These graphs are finally checked by our novel re-sorting procedure (Sect. 9.2.2.2) that compares the two adjacent graphs and uses the previous one as a baseline graph when validating the next one. Note that our sorting also provides another benefit—all duplicated executions (i.e., executions whose memory-access interleaving patterns have been observed by previous executions) can be easily removed during the sorting process.

We treat each execution signature as a multi-word integer value during the sorting process. Thus, our method is sensitive to how we place signature words within the execution signature. From our preliminary experiment, we settled on the following integer layout. We place the first thread's signature in the most significant position, and the last thread's signature in the least significant position. If a per-thread signature consists of multiple words, we place the first word of the per-thread signature in the most significant position within the thread's position mentioned earlier, and the last word in the least significant position. We compared this layout with an alternative layout that places signature words from relevant code sections side by side. We found that this alternative layout created more diverse reads-from relationships between adjacent signatures.

9.2.2.2 Re-sorting Topological Order

MTraceCheck's graph checking examines constraint graphs in a collective manner. Specifically, MTraceCheck verifies each of the constraint graphs, one at a time, in ascending order of their corresponding signatures. The first graph is checked as a

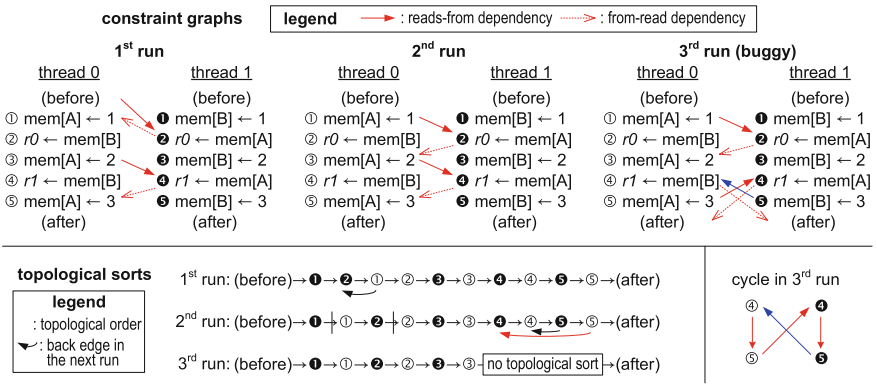

Fig. 9.8 Re-sorting a topologically sorted graph for three test runs. The topological sort from the previous run is partially rearranged for the next run. The sorting boundaries (leading and trailing) are calculated based on backward edges that are introduced by the next run. The absence of topological sort in the segment between the two boundaries indicates a consistency violation

conventional graph checking that topologically sorts all vertices of the graph. Starting from the second graph, MTraceCheck performs a partial re-sorting that applies to only the portion that differs from the prior graph in the sorted order.

MTraceCheck's re-sorting technique uses the result of topological sorting on the previous graph. From the previous topological sort, it re-sorts the region determined by two boundaries: leading boundary and trailing boundary. We decide the two boundaries in a way that the topological sort outside the boundaries can be untouched. To achieve this goal, we decide the leading boundary as the first vertex (in the previous sort) that is connected to a backward edge in the current graph under validation. The trailing boundary is decided in a similar manner; it is the last vertex that is adjacent to a backward edge introduced by the current graph. When deciding leading and trailing boundaries, we consider only new backward edges, but neither forward nor removed edges. In the best case, our re-sorting can skip the entire graph if there is no backward edge.

Figure 9.8 illustrates our re-sorting procedure on three constraint graphs obtained from a 2-threaded program. Suppose that we sorted three execution signatures and reconstructed the three graphs in ascending order of the signatures. MTraceCheck's graph-checking scheme starts with the first graph—this first-time topological sorting is identical to a conventional complete graph checking. MTraceCheck then uses the result of this first-time sorting for the second graph. It examines each of the edges newly introduced in the second graph, ①→❷ and ❷→③, and finds that only the former is backward. This backward edge is illustrated with the backward arrow below the first run's topological sort. From this backward edge, we decide ❷ as the leading boundary and ① as the trailing boundary. The order of the two nodes is then swapped in the topological sort as shown in the second run's diagram. The third run demonstrates a buggy scenario; there is no topological sort of the four vertices

enclosed by the backward edge ⑤→❹. This consistency violation is also illustrated by the cyclic happens-before relationships highlighted in the bottom right of the figure.

9.3 Experimental Evaluation

We evaluated MTraceCheck in two real systems in various test configurations. We first explain our experimental setup in Sect. 9.3.1, then discuss the characteristics of our constrained-random test programs in Sect. 9.3.2. We quantify the benefits and drawbacks of our signature-based memory-tracking method in Sect. 9.3.3 and our collective graph checking in Sect. 9.3.4, respectively.

9.3.1 Experimental Setup

We evaluated MTraceCheck in two different real systems: an x86-based desktop and an ARM-based single-board computer, as summarized in Table 9.1. For each of the systems, we built a bare-metal operating environment that uses no operating system. This bare-metal environment is specialized for our validation tests and performs minimal initialization tasks for caches, MMUs, and page tables. This environment enables an uninterrupted execution of our validation tests with no context switching.

Our bare-metal operating environment initializes each of the systems slightly differently. In our x86 bare-metal environment, the boot-strap processor awakens the secondary cores using interprocessor interrupt (IPI) messages, followed by initializations for caches, page tables, and MMUs. After the initializations, we first allocate test threads in the secondary cores, and then in the primary boot-strap core, but only when no secondary core is available (i.e., 4-threaded tests in our experiments).

Table 9.1 Specifications of the systems under validation

System	x86-64 desktop	ARMv7 single-board computer
Processor	Intel Core 2 Quad Q6600	Samsung Exynos 5422 (big.LITTLE)
MCM	x86-TSO	weakly-ordered memory model
Operating frequency	2.4 GHz	800 MHz (scaled down)
Number of cores	4 (no hyper-threading)	4 (Cortex-A7) + 4 (Cortex-A15)
Cache organization	32+32 KB (L1), 8 MB (L2)	A7: 32+32 KB (L1), 512 KB (L2) A15: 32+32 KB (L1), 2 MB (L2)
Cache configuration	Write back (both L1 and L2)	Write back (L1), write through (L2)

Table 9.2 Test-generation parameters

Test-generation parameter	Values
Number of test threads	2, 4, 7
Number of static memory operations per thread	50, 100, 200
Number of distinct shared memory locations	32, 64, 128

In our ARM bare-metal environment, a primary core in the Cortex-A7 cluster runs the Das U-Boot boot loader [19], which in turn launches our test programs. At the beginning of each test program, the primary core powers up other secondary cores. The secondary cores are then switched to supervisor mode, while the primary core remains in hypervisor mode in order to keep running the boot loader. All caches, page tables, and MMUs are then initialized before the test starts. At the beginning of each test, we allocate test threads to the big cores in the Cortex-A15 cluster then to the little cores in the Cortex-A7 cluster. For example, 4-threaded tests utilize all the four big cores in the Cortex-A15 cluster.

Table 9.2 shows three key parameters used when we generate our constrained-random test programs. We chose these parameter values considering the prior work [20, 24]. We then created 21 representative *test configurations* by combining these parameter values, as shown on the x-axes of the two graphs in Fig. 9.9. The naming convention for the test configuration is as follows: [ISA]-[test threads]-[static memory operations per thread]-[distinct shared memory locations]. For example, ARM-2-50-32 represents a test configuration for the ARM ISA with 2 threads, each issuing 50 memory operations to 32 distinct shared memory locations. For each test configuration, we generated 10 distinct tests with different random seeds and ran each random test 5 times. To remove unintended dependencies across the 5 test runs, we applied a hard reset before starting each test run. Each memory operation is either load or store operation generated with equal probability (i.e., load 50% and store 50%), and transfers a 4-byte data.

Each test run iterates a test-routine loop that encloses the generated memory operations. The repeated execution of the loop is intended to exhibit various memory-access interleaving behaviors. Unless otherwise noted, we iterated this loop 65,536 times. In addition, the beginning of the loop includes a synchronization routine waiting until the previous iteration is completed for all threads, followed by a shared memory initialization and a memory barrier instruction (`mfence` for x86 and `dmb` for ARM). This custom synchronization routine is implemented with a conventional sense-reversal centralized barrier.

9.3.2 Nondeterminism in Memory Ordering

In this section, we quantify the diversity of memory ordering patterns observed from test programs. Figure 9.9 presents the number of unique memory-access interleaving patterns across our 21 test configurations (averaged over 10 tests for each test configuration), measured by counting unique execution signatures. As explained, each multithreaded test program runs 65,536 times in a loop, except for ARM-2-200-32* where its test routine is iterated 1 million times. To summarize the figure, we observe almost no duplicates in several configurations on the top linear-scale graph, while we observe very few distinct interleavings in several test configurations on the bottom logarithmic-scale graph. The dark-red bars in the figure report the results for our baseline address-generation scheme where there is no false sharing among shared memory locations. In those bars, for example, ARM-7-200-64 presents 65,536 unique memory-access interleaving patterns (100%), while ARM-2-50-32 only reveals 11 unique patterns on average (0.02%).

Among the three test-generation parameters of Table 9.2, the number of threads is the most dominant parameter diversifying memory-access interleaving behaviors. For example, we observed about 7 distinct patterns in ARM-2-50-64, 22,124 patterns in ARM-4-50-64, and 65,374 patterns in ARM-7-50-64. Note that the total number of memory operations differs in these three configurations. Under the constant total number of operations, we again observe a significant increase of interleaving patterns: ARM-2-100-64 with 123 patterns versus ARM-4-50-64 with 22,124 patterns.

The number of memory operations is the second influential parameter affecting nondeterministic interleaving behaviors, but this parameter is much less significant than the number of threads. For example, we observed 11 patterns in ARM-2-50-32, 508 patterns in ARM-2-100-32, and 35,679 patterns in ARM-2-200-32. In addition, increasing the number of shared memory locations leads to fewer memory accesses per memory location, thus reducing the number of unique interleaving patterns. ARM-2-200-64 exhibits only 9,638 patterns, much fewer than the 35,679 patterns in ARM-2-200-32.

In most 7-threaded configurations, almost all iterations exhibit very different memory-access patterns. This is partly because of the relatively low loop-iteration count that we used (65,536). As a sensitivity study, we evaluate the impact of iteration count for ARM-2-200-32, where we compared the results from two iteration counts: 35,679 unique patterns out of 65,536 (54%) versus 311,512 unique patterns out of 1,048,576 (30%), as shown in the two left-most bars in the linear-scale graph.

Moreover, we notice that our two systems show different levels of memory-access interleaving. Specifically, our x86-based system exhibits fewer distinct interleaving behaviors, compared to our ARM-based system. This is partly because of the stricter memory ordering rules enforced by the system's TSO model. To follow the TSO model, the x86-based system needs to restrict some memory re-ordering behaviors, while such re-ordering behaviors are allowed by the ARM-based system. We believe that the memory model is the major contributor to the difference between the two systems, among other factors such as load-store queue (LSQ) size, cache organiza-

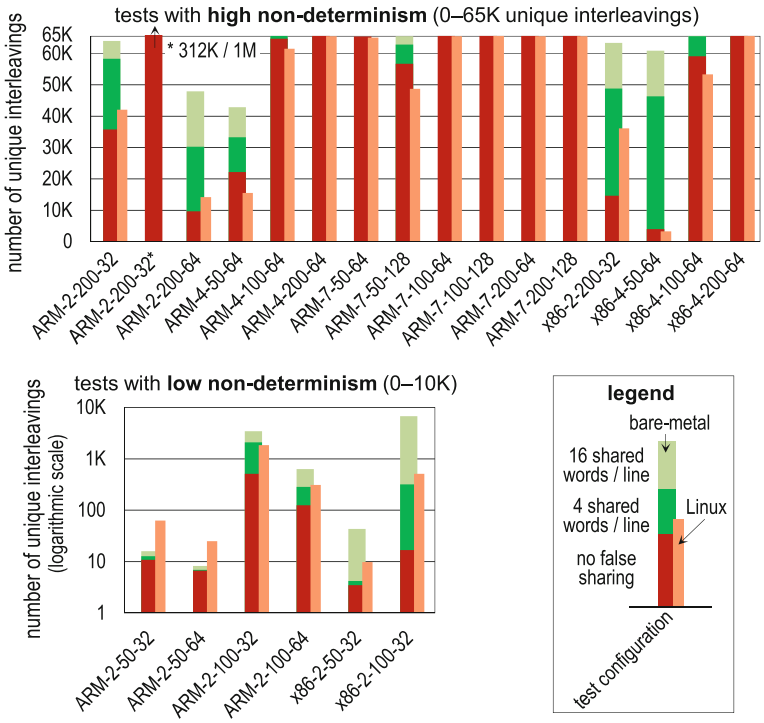

Fig. 9.9 The number of unique memory-access interleaving patterns. For each multithreaded test, we measured the number of unique memory-access interleaving signatures from 65,536 repeated runs. Most of the 2-threaded tests exhibit only a handful of distinct interleavings as shown in the logarithmic-scale graph on the bottom, while long 7-threaded tests show unique interleavings in almost all iterations as shown in the linear-scale graph on the top. False sharing of cache lines contributes to diversifying interleavings, as shown in the green and light-green bars. Compared to our bare-metal operating environment, the Linux environment creates more unique interleavings in 2-threaded tests, while it restricts interleavings in 4-threaded and 7-threaded tests

tion, interconnect, etc. For a fair comparison, each x86 test is generated in a way that it has the identical memory-access patterns as in the ARM counterpart.

Impact of False Sharing of Cache Lines

Modern multi-core systems provide coherent caches among cores. The memory consistency in these systems is implemented in consideration of the underlying cache coherence mechanism. In cache-coherent systems, a cache line is the minimal granularity of a data transfer among caches. Our prior explanation refers to the dark-red bars in Fig. 9.9 that are obtained from tests with no false sharing. In other words, only one shared word is placed in each cache line, while the rest of the cache line is not accessed at all.

Placing multiple shared words in a cache line creates additional contention among threads, and diversifies memory-access interleaving behaviors. The green and light-

green bars in Fig. 9.9 present the numbers of unique memory-access interleaving patterns for two different data layouts; 4 and 16 shared words per cache line, respectively. By the false sharing, x86-4-50-64 shows the most dramatic increase among others: from 3,964 (no false sharing) to 46,266 (4 shared words) to 60,868 (16 shared words) unique patterns. We observe a larger increase in the x86-based system than in the ARM-based system.

Impact of the Operating System

In our previous discussion, we exclusively used bare-metal operating environments explained earlier. Besides the bare-metal environments, our test programs can be targeted to the Linux operating system, which allows context switching and interferences with other concurrent tasks sharing the memory subsystem. The operating system also introduces another source of test perturbation: the OS can shuffle the memory addresses by paging, while our bare-metal environment has a direct mapping between virtual and physical addresses. To quantify the perturbation of an OS, we re-targeted the same set of tests to the following Linux systems: Ubuntu MATE 16.04 for the ARM-based system and Ubuntu 10.04 LTS for the x86-based system. Under these Ubuntu systems, test threads are launched via the m5threads pthreads library [14], although, after the launch, synchronizations among test threads are carried out by our own synchronization primitives as in our bare-metal environments.

The light-red bars in Fig. 9.9 show our experimental results from the Linux environment with no false sharing. We observe two noticeable trends. In 2-threaded tests, the number of unique interleavings increases compared to the bare-metal counterparts. We believe that fine-grained (i.e., instruction-level) interferences dominate in these tests, thus diversifying the interleavings. For example, OS system tasks (which are unrelated to test threads) can perform their memory operations that can perturb the timing behavior of test threads' memory operations. In both 4-threaded and 7-threaded tests, on the contrary, the opposite trend holds. We believe that this tendency comes from coarse-grained (i.e., thread-level) interferences. For instance, some of the test threads can be preempted by OS, and then resume running after all other threads are already done, in which case this stranded thread is subject to experience much fewer memory-access interleavings.

9.3.3 Validation Performance

We measured two major components of validation time: the time spent on result-checking and the time spent on test execution. In summary, MTraceCheck achieves a significant speedup in result-checking at the expense of a small runtime increase in test execution. We provide detailed experimental results and insights below.

Result-Checking Speedup

We performed a constraint-graph checking on a host machine equipped with an Intel Core i7 860 2.8 GHz and 8 GB of main memory, running Ubuntu 16.04 LTS. This host machine is more powerful than our systems under validation (Table 9.1) because

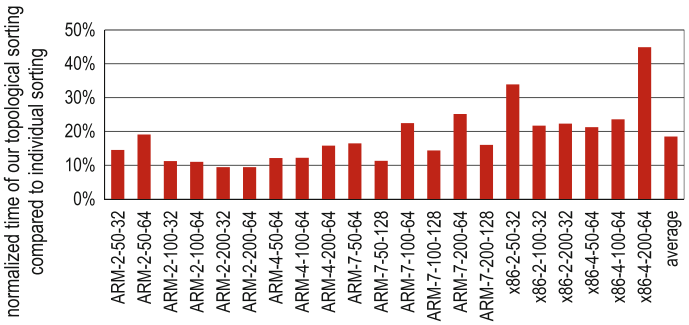

Fig. 9.10 Normalized violation-checking computation. The collective graph checking reduces topological sorting time by 81% on average, compared to a conventional individual-graph checking

the result checking takes significantly more time than the test execution. For reproducibility of our results, we adopted a well-known topological sort program, `tsort` that is part of GNU core utilities [23]. We modified the original `tsort` program so as to efficiently check multiple constraint graphs generated from a test. Specifically, for both the baseline individual checking and MTraceCheck's collective checking, vertex data structures are recycled for all constraint graphs, while edge data structures are not. We excluded the time spent in reading input graph files, assuming that all graphs are loaded into memory beforehand. We ran the `tsort` program 5 times for each evaluation to mitigate random interferences from the host machine's Linux environment. We used the signatures obtained in the Linux environment (the light-red bars in Fig. 9.9) instead of the signatures obtained in the bare-metal environment. For a fair comparison, we considered only unique constraint graphs for both checking techniques.

Figure 9.10 plots the time spent on topological sorting in our collective graph-checking method, normalized to a conventional baseline approach that examines each constraint graph separately. We observe that our collective checking consistently reduces the overall computation by 81% on average. In addition, we observe a noticeable difference between our ARM and x86 platforms: the benefit of our technique is smaller in the x86 platform. This difference is also shown in the wall-clock time spent on our topological sorting, which is not shown on the graph. For the graphs obtained from the ARM platform, the wall-clock time ranges from 2.4 μs for ARM-2-50-64 to 1.2 s for ARM-7-200-64. The wall-clock time for the x86 platform's graphs ranges from 8.6 μs for x86-2-50-32 to 4.6 s for x86-4-200-64.

We conducted an in-depth analysis to investigate the difference between the ARM and x86 platforms. From our analysis, we found that two major factors that contribute to the speedup of our collective graph checking. The first speedup factor we notice is that many constraint graphs can be validated immediately without modifying the previous topological sort. This situation is highlighted especially in the left-most bar (ARM) of Fig. 9.11, where most constraint graphs except for the first graph do not require any re-sorting. This is because the `tsort` program unwittingly places store

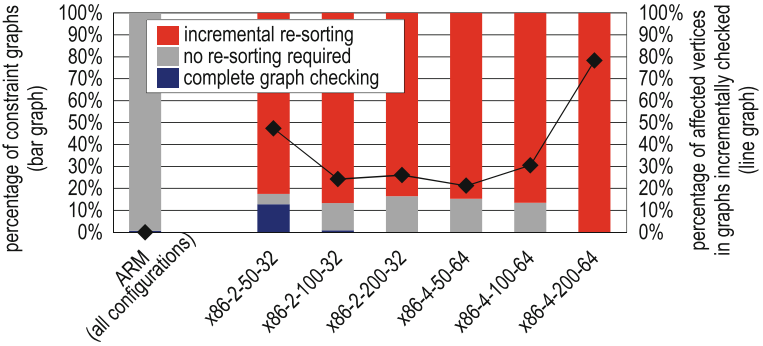

Fig. 9.11 Sources of speedup in our collective graph checking. Most graphs from ARM tests are skipped and do not require any topological re-sorting. A majority of graphs from x86 tests needs to be re-sorted partially; the re-sorting affects up to 78% of the vertices

operations prior to load operations since stores do not depend on any load operations in the absence of fence operations, which is the case for our ARM tests. Even in this case, however, our collective checking still needs to check every new edge whether the edge is backward or not, because the program is MCM-agnostic and not aware of the fact that no backward edge exists before the program examines all of the new edges. Consequently, this edge checking is the major computation performed while checking the ARM test results.

Second, for our x86 tests, we also observe that a small portion of constraint graphs, up to 16% of the graphs, can be validated without re-sorting. But a majority of the x86 graphs needs incremental re-sorting as shown in the red segments in the figure, which ranges from 82% in x86-2-50-32 to almost 100% in x86-4-200-64. For these graphs, we compute the percentage of the vertices affected by re-sorting, as shown in the line plotted in the figure. This percentage ranges from 21% in x86-4-50-64 to 78% in x86-4-200-64. This x86-4-200-64 test configuration has the largest re-sorting portion with respect to both of the metrics: the percentage of re-sorted graphs and the percentage of affected vertices. Thus this test configuration gains the least benefit from our collective checking as shown in Fig. 9.10.

Test Execution Overhead

We report MTraceCheck's runtime overheads, measured in our ARM bare-metal system. We used built-in hardware performance monitors available in the ARM system [10] to accurately measure the runtime. Figure 9.12 summarizes two types of runtime overheads introduced by MTraceCheck: (1) signature computation (i.e., the time spent in running chains of branch and arithmetic operations and storing signatures) and (2) execution signature sorting. We implemented our signature-sorting routine using a balanced binary tree written in the C programming language, and ran

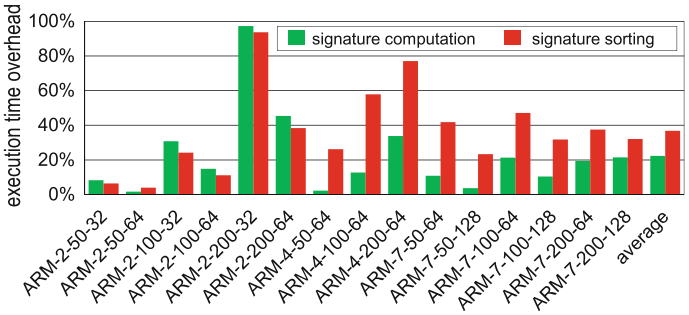

Fig. 9.12 Test execution overhead. The test runtime is increased by 22% on average due to our instrumented code for signature computation. In addition, signature sorting also increases the runtime by 38% on average

this sorting routine on the primary core in the Cortex-A7 cluster[1] after all repeated test executions completed.

The original test takes 0.09–1.1 s to run 65,536 iterations. On top of this baseline, MTraceCheck's signature computation incurs 22% additional time and its signature sorting adds 38% more time on average. Specifically, the signature computation overhead ranges from as low as 1.5% in ARM-2-50-64 to up to 97.8% in ARM-2-200-32, which is an exceptional case. The lowest overhead in ARM-2-50-64 can be explained from the fact that there are only about 7 unique interleaving patterns among 65,536 iterations, thus the branch predictor can almost perfectly predict the directions of the branch operations in our instrumented code. On the contrary, ARM-2-200-32 exhibits a unique interleaving pattern for almost every iteration, thus the branch misprediction penalty becomes much noticeable. The signature-sorting overhead ranges from 3.9% in ARM-2-50-64 to 93.5% in ARM-2-200-32. For the 1 million-iteration version of ARM-2-200-32, we found a 140% signature-sorting overhead, which is not shown in the graph.

9.3.4 Intrusiveness of Code Instrumentation

To quantify the intrusiveness of MTraceCheck's code instrumentation, we first measure the additional memory accesses introduced to track the order among memory operations. These additional memory accesses are not part of the original test and only needed for the validation purpose. As explained in Sect. 9.2.1, reducing the additional memory accesses is a key challenge while tracking the results of memory operations in post-silicon memory consistency tests.

[1] The signature-sorting time can be significantly reduced by two optimizations: (1) the sorting routine can utilize multiple cores available, and (2) the sorting routine can be run on the more powerful Cortex-A15 cluster. These two are not implemented in our experimental evaluations.

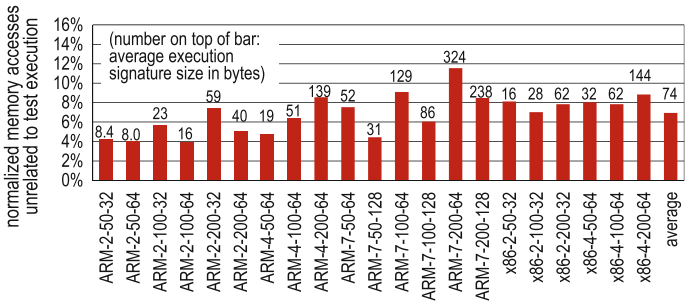

Fig. 9.13 Intrusiveness of verification. Our signature-based tracking greatly reduces memory accesses unrelated to test execution compared to a prior register-flushing approach

We report the amount of these additional accesses in Fig. 9.13, normalized to a conventional register-flushing technique [24] where all loaded values must be flushed back to memory. Compared to the conventional technique, our signature-based technique requires only 7% additional memory accesses on average. The amount of additional memory accesses ranges from 3.9% in ARM-2-100-64 to 11.5% in ARM-7-200-64. Tests with more contention (i.e., more threads, more memory operations, and fewer shared memory locations) have a larger signature footprint, which in turn require more data to be transferred.

The average size of an execution signature is shown on the top of each bar in the figure. Under low-contention tests, the signature size is bounded by the register bit width. In the tests with 2 threads, 50 operations and 32 locations (ARM-2-50-32 and x86-2-50-32), the per-thread signature barely exceeds 32 bits, as explained earlier in Sect. 9.2.1.3. However, we use all the bits of a register to store a per-thread signature: 64 bits for our x86 system and 32 bits for our ARM system. The register-width difference results in a significant difference (almost twice) in the average size of an execution signature between the two systems: 16 bytes for the x86 system and 8.4 bytes for the ARM system. Under high-contention tests, the gap between 32-bit and 64-bit registers becomes narrow, as the per-thread signature indeed requires more than one words. Among the test configurations we tested, ARM-7-200-64 has the largest signature; its size is 324 bytes on average.

In addition, our code instrumentation increases the code size, which is another source of perturbation introduced for the purpose of verification. To quantify this aspect, we measured the size of the instrumented code compared to the original test routine that excludes initialization and signature sorting routines, illustrated in Fig. 9.14. The ratio of the code size ranges from 1.95 in ARM-2-50-64 to 8.16 in ARM-7-200-64. On the top of each bar in the figure, we show the aggregated instrumented code size for all threads. While this increase is fairly significant, the code is still small enough to be accommodated in a processor's L1 instruction cache, which is usually tens of KB in modern processors.

9.4 Bug-Injection Case Studies

We conducted bug-injection experiments using the gem5 full-system simulator [14]. From our bug-injection experiments, we evaluate the intrusiveness of our code instrumentation in a qualitative manner whether or not the instrumented test is still capable of triggering subtle bugs in the full-system simulator. We chose three real bugs, which have been recently reported in prior work [20, 30, 34], and injected each of these bugs one at a time. Before we injected these bugs, we confirmed that these bugs have been fixed in the recent stable version of the gem5 simulator (tag `stable_2015_09_03`). To recreate each of the bugs, we searched the relevant bug fix from the gem5 repository [22] and reverted the fix.

Our gem5 is configured to simulate eight out-of-order x86 cores connected with a 4×2 mesh network. The cache coherence is maintained with a directory-based MESI cache coherence protocol where the directories are located on the four mesh corners. For bugs 1 and 3, we purposefully calibrated the size and the associativity of the L1 data cache (1 KB with 2-way associativity) to intensify the effect of cache evictions under a small working set used by our constrained-random tests, while the rest of the cache configuration is modeled after a recent version of the Intel Core i7 processor. We compiled our tests with the m5threads library as in our Linux environment explained in Sect. 9.3.2, and used the gem5's syscall emulation mode to run our tests.

9.4.1 Bug Descriptions

Bug 1 – An improper cache-invalidation handling leads to load→load violation: This bug is modeled after "MESI,LQ+SM,Inv" presented in [20]. It was fixed in June 2015. It is a variant of the Peekaboo problem [46], where a load operation is speculatively performed earlier than its preceding load operation in the TSO memory model. It is triggered very rarely when a cache receives an invalidation message for

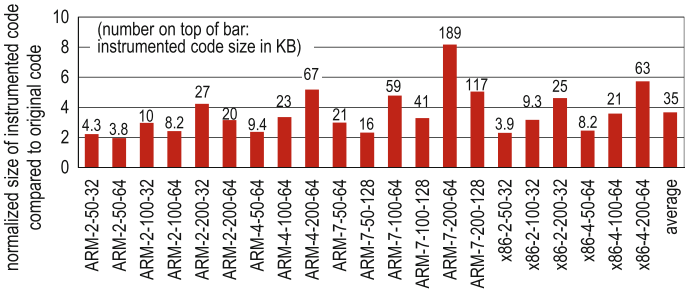

Fig. 9.14 Normalized code size. Our code instrumentation increases the code size by 3.7 times on average

a cache line that is currently transitioning from shared state to modified state (i.e., the cache line is in shared-to-modified transient state). When the cache receives the invalidation message, subsequent load operations to this cache line must be replayed after handling the invalidation message. However, due to this bug, the cache does not squash the subsequent load operations, resulting in a load→load violation.

Bug 2 – Lack of LSQ replay leads to load→load violation: This bug is reported by two independent works: [34] and [20]. It was fixed in March 2014. It manifests in a similar way as bug 1 but is caused by a buggy LSQ that does not invalidate subsequent loads when it receives an invalidation message.

Bug 3 – A race condition in cache coherence protocol: This bug is modeled after the "MESI bug 1" detailed in [30], which is evaluated also in [20]. It was fixed in January 2011. It is triggered when a race condition happens between an L1 writeback message (PUTX) and a write-request message (GETX) from another L1 cache.

9.4.2 Bug-Detection Results

For each of the three bugs, we deliberately chose the test configuration in the second column of Table 9.3 after analyzing the bug description, and generated 101 different constrained-random tests under the same test configuration. The iteration count was set to 1,024, which is greatly reduced from 65,536 in Sect. 9.3, because the gem5 simulation is much slower than the native execution of the systems in Table 9.1.

With MTraceCheck, we were able to successfully detect all the injected bugs. The third column of Table 9.3 summarizes our bug-detection results: how many tests and signatures reveal the bugs. Among the three bugs, bug 1 was the most difficult to find. It was detected by only one test, which generated 29 invalid signatures. We also tried other constrained-random tests and hand-crafted litmus tests for a few days, but these tests failed to trigger the bug. Bug 2 manifested itself in 11 tests. Among the 11 tests, one test generated two incorrect signatures, while each of the other tests generated one incorrect signature. While these two bugs are subtle and appeared only in some tests and iterations, bug 3 caused much more significant failures. For all the 101 tests, the gem5 simulations ended prematurely with internal errors (a protocol deadlock and an invalid transition) from the gem5's ruby memory subsystem.

Table 9.3 Bug-detection results

Bug	Test configuration	Detection results
1	x86-4-50-8 with 4 shared words per cache line	1 test, 29 signatures
2	x86-7-200-32 with 16 shared words per cache line	11 tests, 12 signatures
3	x86-7-200-64 with 4 shared words per cache line	all 101 tests, crash

Figure 9.15 shows a code snippet of the test that exposed bug 1. The first 20 memory operations of thread 0 are omitted for simplicity. In thread 0, there is no store operation to memory address `0x1` until the starred instruction ★, although the omitted code includes store operations to the other addresses in the same cache line (i.e., memory addresses `0x0`, `0x2` and `0x3`). Each of threads 1 and 2 performs a store operation to `0x1` as shown in the figure (gray bold operations). Thread 3 executes three consecutive load operations from memory address `0x1`. Among these three load operations, the first and the third load operations read the initial value of the memory location, while the second reads the value from the store operation ★. From the second and third loaded values, we identify a cyclic happens-before relationship illustrated with blue arrows in the figure: ★ happens before ② (a reads-from relationship), which should happen before ③ (a load→load ordering), which in turn happens before ★ (a from-read relationship).

9.5 Discussions

Reducing the signature size by pruning invalid memory-access interleaving patterns We make a conservative assumption in our code instrumentation (Sect. 9.2.1) to support a wide range of MCMs in a unified framework. In our assumption, each memory operation can be independently interleaved, regardless of memory ordering rules required by MCMs. This conservative assumption leads to two major overheads discussed earlier in Sect. 9.3.4: the increased signature size (Fig. 9.13) and code size (Fig. 9.14). To address these two overheads, we can leverage micro-architectural information during the code instrumentation (*static pruning*). For instance, we can compute a window of outstanding memory operations (as in [45]) by taking into account the number of LSQ entries, etc. By doing so, we can prune infeasible options for loaded values. Unfortunately, we could not gather sufficient micro-architectural details to apply this static pruning optimization in our real-system evaluations.

Another possible optimization relies on a runtime technique (*dynamic pruning*) computing a frontier of memory operations in strong MCMs (e.g., TSO). In these MCMs, reading from memory operations older than the frontier is often considered

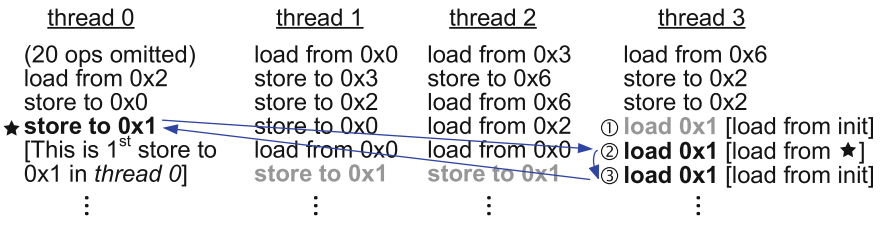

Fig. 9.15 A load→load ordering violation detected by a 4-threaded test. In thread 3, loads ② and ③ are incorrectly re-ordered due to bug 1

to be invalid. To implement this optimization, each thread needs to track the set of recent store operations performed. This dynamic pruning technique, however, can complicate the signature computation and decoding processes.

Improving scalability As the size of the test program grows, both the signature size and the instrumented code size also increase. While we did not experience scalability issues for the test configurations we tested, it is possible that our code instrumentation can be overly intrusive for a large test program with many memory-access interleaving opportunities. To improve the scalability, we can merge multiple independent code segments to create a single test program (as in [42]). The independent code segments share cache lines but do not share memory addresses within the cache lines, allowing only false sharing across the code segments.

Store atomicity We make no assumption on store atomicity (single-copy atomic, multiple-copy atomic, or non-multiple-copy atomic) in Sect. 9.2. However, we did not evaluate MTraceCheck in a single-copy atomic real system except for the limit study presented in Sect. 9.2.2.1. Constraint graphs for a single-copy atomic system contain additional dependency edges [12, 35] that are not necessarily included in a multiple-copy or non-multiple-copy atomic system, which may decrease the benefit of our collective checking due to larger re-sorting windows.

9.6 Conclusions

Multi-core processors experience a variety of different memory-access interleaving behaviors while they run multithreaded programs. Verifying such nondeterministic behaviors involves analyzing many different memory orders, if these orders abide by the memory consistency model. Post-silicon memory consistency validation aims at checking very subtle memory ordering behaviors that are rarely observed. To improve the efficiency of memory consistency validation, we propose MTraceCheck that tackles two major obstacles: limited observability and heavy result-checking computation. The first obstacle is mitigated by our novel signature-based memory-tracking method, which instruments observability-enhancing code. The instrumented code computes a compact signature value representing memory-access interleaving patterns observed at runtime. It also minimally perturbs the memory accesses in the original test, thus it is still capable of triggering subtle memory consistency bugs. The second obstacle is alleviated by our collective graph-checking algorithm that leverages the structural similarities among constraint graphs. Our comprehensive evaluations show that MTraceCheck can quickly detect subtle memory consistency bugs.

The MTraceCheck source code is available on our GitHub repository: https://github.com/leedoowon/MTraceCheck. The repository includes the constrained-random test generator for memory consistency (Sect. 9.3.1), the code instrumentation script (Sect. 9.2.1), the architectural simulator (Sect. 9.2.2.1), and the collective graph

checker (Sect. 9.2.2.2). The current version supports three ISAs: x86-64, ARMv7, and RISC-V.

Acknowledgements We would like to thank Prof. Todd Austin, Biruk Mammo, and Cao Gao for their advice and counseling throughout the development of this project. The work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Doowon Lee was also supported by a Rackham Predoctoral Fellowship at the University of Michigan.

References

1. A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, A. Ziv, Threadmill: a post-silicon exerciser for multi-threaded processors, in *Proceedings of the 48th Design Automation Conference* (2011), pp. 860–865. https://doi.org/10.1145/2024724.2024916
2. S.V. Adve, K. Gharachorloo, Shared memory consistency models: a tutorial. Computer **29**(12), 66–76 (1996). https://doi.org/10.1109/2.546611
3. J. Alglave, A formal hierarchy of weak memory models. Form. Methods Syst. Design **41**(2), 178–210 (2012). https://doi.org/10.1007/s10703-012-0161-5
4. J. Alglave, L. Maranget, S. Sarkar, P. Sewell, Fences in weak memory models, in *Computer Aided Verification: 22nd International Conference, CAV 2010* (2010), pp. 258–272. https://doi.org/10.1007/978-3-642-14295-6_25
5. J. Alglave, L. Maranget, S. Sarkar, P. Sewell, Litmus: running tests against hardware, in *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011* (2011), pp. 41–44. https://doi.org/10.1007/978-3-642-19835-9_5
6. J. Alglave, L. Maranget, M. Tautschnig, Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014). https://doi.org/10.1145/2627752
7. ARM: Barrier Litmus Tests and Cookbook (2009)
8. ARM: Cortex-A9 MPCore Programmer Advice Notice Read-after-Read Hazards, ARM Reference 761319 (2011)
9. ARM: Embedded Trace Macrocell Architecture Specification (2011)
10. ARM: ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (2012)
11. ARM: ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile (2017)
12. Arvind, J.W. Maessen, Memory model = instruction reordering + store atomicity, in *Proceedings of the 33rd Annual International Symposium on Computer Architecture* (2006), pp. 29–40. https://doi.org/10.1109/ISCA.2006.26
13. T. Ball, J.R. Larus, Efficient path profiling, in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (1996), pp. 46–57. https://doi.org/10.1109/MICRO.1996.566449
14. N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, D.A. Wood, The gem5 simulator. ACM SIGARCH Comput. Archit. News **39**(2), 1–7 (2011). https://doi.org/10.1145/2024716.2024718
15. H.W. Cain, M.H. Lipasti, R. Nair, Constraint graph analysis of multithreaded programs, in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* (2003), pp. 4–14. https://doi.org/10.1109/PACT.2003.1237997
16. K. Chen, S. Malik, P. Patra, Runtime validation of memory ordering using constraint graph checking, in *2008 IEEE 14th International Symposium on High Performance Computer Architecture* (2008), pp. 415–426. https://doi.org/10.1109/HPCA.2008.4658657

17. Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, H. Pan, Fast complete memory consistency verification, in *2009 IEEE 15th International Symposium on High Performance Computer Architecture* (2009), pp. 381–392. https://doi.org/10.1109/HPCA.2009.4798276
18. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd edn. (The MIT Press, Cambridge, 2009)
19. Das U-Boot – the universal boot loader (2016), http://www.denx.de/wiki/U-Boot
20. M. Elver, V. Nagarajan, McVerSi: a test generation framework for fast memory consistency verification in simulation, in *2016 IEEE International Symposium on High Performance Computer Architecture* (2016), pp. 618–630. https://doi.org/10.1109/HPCA.2016.7446099
21. N. Foutris, D. Gizopoulos, M. Psarakis, X. Vera, A. Gonzalez, Accelerating microprocessor silicon validation by exposing ISA diversity, in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), pp. 386–397. https://doi.org/10.1145/2155620.2155666
22. gem5 mercurial repository host (2016), http://repo.gem5.org
23. GNU coreutils version 8.25 (2016), http://ftp.gnu.org/gnu/coreutils
24. S. Hangal, D. Vahia, C. Manovit, J.Y.J. Lu, S. Narayanan, TSOtool: a program for verifying memory systems using the memory consistency model, in *Proceedings of the 31st Annual International Symposium on Computer Architecture* (2004), pp. 114–123. https://doi.org/10.1109/ISCA.2004.1310768
25. IBM: Power ISA Version 2.07B (2015)
26. Intel: Intel 64 Architecture Memory Ordering White Paper (2007)
27. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual (2015)
28. Intel: 6th Generation Intel Processor Family Specification Update (2016)
29. k-medoids algorithm (2016), https://en.wikipedia.org/wiki/K-medoids
30. R. Komuravelli, S.V. Adve, C.T. Chou, Revisiting the complexity of hardware cache coherence and some implications. ACM Trans. Archit. Code Optim. **11**(4), 37:1–37:22 (2014). https://doi.org/10.1145/2663345
31. L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
32. D. Lee, V. Bertacco, MTraceCheck: validating non-deterministic behavior of memory consistency models in post-silicon validation, in *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), pp. 201–213. https://doi.org/10.1145/3079856.3080235
33. D. Lin, T. Hong, Y. Li, E. S, S. Kumar, F. Fallah, N. Hakim, D.S. Gardner, S. Mitra, Effective post-silicon validation of system-on-chips using quick error detection. IEEE Trans. Comput. Aided Design Integr. Circuits Syst. **33**(10), 1573–1590 (2014). https://doi.org/10.1109/TCAD.2014.2334301
34. D. Lustig, M. Pellauer, M. Martonosi, PipeCheck: specifying and verifying microarchitectural enforcement of memory consistency models, in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), pp. 635–646. https://doi.org/10.1109/MICRO.2014.38
35. D. Lustig, C. Trippel, M. Pellauer, M. Martonosi, ArMOR: defending against memory consistency model mismatches in heterogeneous architectures, in *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), pp. 388–400. https://doi.org/10.1145/2749469.2750378
36. D. Lustig, A. Wright, A. Papakonstantinou, O. Giroux, Automated synthesis of comprehensive memory model litmus test suites, in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), pp. 661–675. https://doi.org/10.1145/3037697.3037723
37. S. Mador-Haim, R. Alur, M.M. Martin, Generating litmus tests for contrasting memory consistency models, in *Computer Aided Verification: 22nd International Conference, CAV 2010* (2010), pp. 273–287. https://doi.org/10.1007/978-3-642-14295-6_26

38. B.W. Mammo, V. Bertacco, A. DeOrio, I. Wagner, Post-silicon validation of multiprocessor memory consistency. IEEE Trans. Comput. Aided Design Integr. Circuits Syst. **34**(6), 1027–1037 (2015). https://doi.org/10.1109/TCAD.2015.2402171
39. Y.A. Manerkar, D. Lustig, M. Martonosi, M. Pellauer, RTLCheck: verifying the memory consistency of RTL designs, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017), pp. 463–476. https://doi.org/10.1145/3123939.3124536
40. A. Meixner, D.J. Sorin, Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. IEEE Trans. Dependable Secur. Comput. **6**(1), 18–31 (2009). https://doi.org/10.1109/TDSC.2007.70243
41. M. Naylor, S.W. Moore, A. Mujumdar, A consistency checker for memory subsystem traces, in *2016 Formal Methods in Computer-Aided Design (FMCAD)* (2016), pp. 133–140. https://doi.org/10.1109/FMCAD.2016.7886671
42. T. Rabetti, R. Morad, A. Goryachev, W. Kadry, R.D. Peterson, SLAM: SLice And Merge - effective test generation for large systems, in *Hardware and Software: Verification and Testing* (2013), pp. 151–165
43. A. Roy, S. Zeisset, C.J. Fleckenstein, J.C. Huang, Fast and generalized polynomial time memory consistency verification, in *Computer Aided Verification: 18th International Conference, CAV 2006* (2006), pp. 503–516. https://doi.org/10.1007/11817963_46
44. E. Seligman, T. Schubert, M.V.A.K. Kumar, *Formal Verification* (Morgan Kaufmann, San Francisco, 2015)
45. O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, M. Horowitz, Verification of chip multiprocessor memory systems using a relaxed scoreboard, in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (2008), pp. 294–305. https://doi.org/10.1109/MICRO.2008.4771799
46. D.J. Sorin, M.D. Hill, D.A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 1st edn. (Morgan & Claypool Publishers, 2011)
47. The Coq proof assistant (2016), https://coq.inria.fr
48. I. Wagner, V. Bertacco, Reversi: Post-silicon validation system for modern microprocessors, in *IEEE International Conference on Computer Design* (2008), pp. 307–314. https://doi.org/10.1109/ICCD.2008.4751878
49. A. Waterman, K. Asanovic, SiFive Inc., The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2 (2017)
50. D. Weaver, T. Germond, *The SPARC Architectural Manual (Version 9)* (Prentice-Hall Inc., Englewood Cliffs, 1994)