MTraceCheck: Validating Non-Deterministic Behavior of Memory Consistency Models in Post-Silicon Validation

Doowon Lee
University of Michigan
doowon@umich.edu

Valeria Bertacco
University of Michigan
valeria@umich.edu

ABSTRACT

This work presents a minimally-intrusive, high-performance, post-silicon validation framework for validating memory consistency in multi-core systems. Our framework generates constrained-random tests that are instrumented with observability-enhancing code for memory consistency verification. For each test, we generate a set of compact signatures reflecting the memory-ordering patterns observed over many executions of the test, with each of the signatures corresponding to a unique memory-ordering pattern. We then leverage an efficient and novel analysis to quickly determine if the observed execution patterns represented by each unique signature abide by the memory consistency model. Our analysis derives its efficiency by exploiting the structural similarities among the patterns observed.

We evaluated our framework, MTraceCheck, on two platforms: an x86-based desktop and an ARM-based SoC platform, both running multi-threaded test programs in a bare-metal environment. We show that MTraceCheck reduces the perturbation introduced by the memory-ordering monitoring activity by 93% on average, compared to a baseline register flushing approach that saves the register's state after each load operation. We also reduce the computation requirements of our consistency checking analysis by 81% on average, compared to a conventional topological sorting solution. We finally demonstrate the effectiveness of MTraceCheck on buggy designs, by evaluating multiple case studies where it successfully exposes subtle bugs in a full-system simulation environment.

CCS CONCEPTS

• **Computer systems organization** → *Multicore architectures*; • **Hardware** → *Bug detection, localization and diagnosis*;

KEYWORDS

memory consistency model, post-silicon validation

ACM Reference format:
Doowon Lee and Valeria Bertacco. 2017. MTraceCheck: Validating Non-Deterministic Behavior of Memory Consistency Models in Post-Silicon Validation. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080235

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '17, June 24-28, 2017, Toronto, ON, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080235

1 INTRODUCTION

Over the past decade, microprocessor designs have been integrating an increasing number of cores in a single chip, to keep power consumption in check while maximizing computing capability. Indeed, individual cores in a many-core system are tailored for energy efficiency, significantly improving throughput per watt by exploiting thread-level parallelism. To take advantage of this parallelism, cores can be programmed to collectively contribute to advance a single application, thus offering higher energy efficiency than a high-frequency single-core system with the same power budget. For instance, the Oracle SPARC M7 chip [30] includes 32 cores and can execute 256 threads in a single chip, enabling massive thread-level parallelism for parallel applications.

Many-core systems are commonly equipped with large caches and memory units integrated alongside the cores. All threads participating in a multi-threaded application share a common memory space, and, for the most part, they communicate with each other via shared memory. In the absence of inter-thread synchronizations, each thread runs its own program code without being subject to interruptions by other threads, and thus accesses to the shared memory may be arbitrarily interleaved. This mainstream setup leads to non-deterministic outcomes, posing a difficult challenge to microprocessor verification engineers, who validate a test execution by comparison with its correct outcome.

During post-silicon microprocessor validation, the primary activity is to detect rarely occurring bugs, triggered by complex corner cases, deep down in a design's state space. It is thus crucial to run as many tests as possible in the limited time available before product release, so as to expose as many of those rare bugs as possible. Fortunately, post-silicon validation platforms operate at a chip's native speed, generating a vast number of test results on the fly. However, this high production rate requires the validation methodology to keep the pace with test output generation. Moreover, in this setup, only a handful of internal signals can be observed at runtime [1, 27, 39]; hence, verifying a system in the presence of variable, non-deterministic memory-access interleavings becomes even more challenging.

To address the latter challenge, some modern processors include trace buffers, such as Intel's Branch Trace Store (BTS) [25] or ARM's Embedded Trace Macrocell (ETM) [8]: they can trace branches, interrupts and exceptions in one case and instructions and data in the other. However, these tracing structures are often limited and unfit in size for memory-ordering validation tasks,

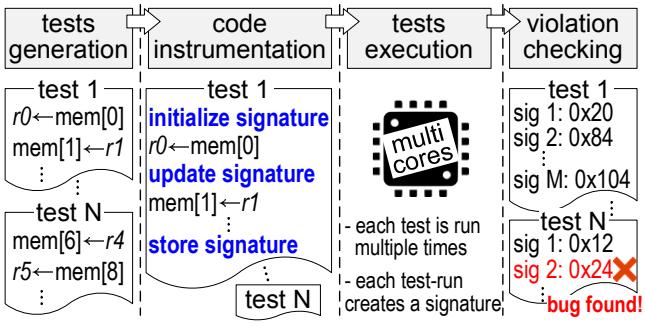

Figure 1: MTraceCheck overview. Multi-threaded constrained-random tests are generated and then augmented with our observability-enhancing code, which generates a memory-access interleaving signature at runtime. After multiple test runs, signatures are collectively checked by exploiting their similarities, accelerating the validation process.

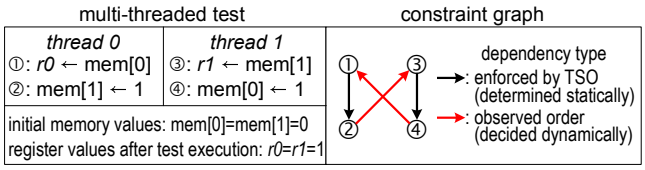

Figure 2: A two-threaded test execution and its constraint graph. Among various test results, *r0*=*r1*=1 is considered invalid in the TSO model. Correspondingly, its constraint graph identifies a cycle.

which require logging and checking the outcome of each memory operation. To address the heavy computation required for checking test outputs, further exacerbated by the potentially high non-determinism of memory ordering, researchers have proposed solutions to enhance the observability of memory access orderings by either introducing additional hardware or through intrusive software modifications of the application's code. In the former category, [14, 35], for instance, propose to modify the memory subsystem's hardware for the sole purpose of validation. As an example of the latter approach, [24] performs additional memory accesses to record the results of the application's memory operations. These additional accesses, in turn, may alter the execution flow of the original test programs (*i.e.*, they are intrusive). Moreover, when memory access traces are checked in a host machine, the amount of data to be transferred to the host can be extremely large, affecting the performance of the overall validation process. In addressing these problems, we strive to minimize intrusiveness in gathering insights on each specific test execution flow and drastically reduce the computational overheads in validating test outcomes, thus greatly boosting the number of tests that can be run in post-silicon validation.

In this paper, we propose a post-silicon memory-ordering validation framework, called **MTraceCheck**, that efficiently checks non-deterministic interleavings of memory accesses in multi-core systems. Figure 1 summarizes the MTraceCheck validation flow in four steps: constrained-random tests generation, code instrumentation, tests execution, and violation checking. Firstly, our validation flow generates multi-threaded tests to stimulate rare memory-access interleavings. The generated tests are then augmented with our observability-enhancing code, which computes a compact signature, corresponding to the observed memory-access interleaving for a given test execution (Section 3). After tests are run multiple times in a post-silicon validation platform, we collect and classify all the signatures generated. Signatures are collectively checked by our novel analysis framework, which detects and exploits similarity among distinct memory-access interleavings, thus avoiding repetitive, duplicate checks (Section 4). In this paper, we make the following contributions:

- We introduce a novel code-instrumentation technique to enhance the observability of memory-access interleavings in post-silicon multi-core validation. Our technique is minimally intrusive; it reduces by 93% the memory operations unrelated to the test execution, compared to a conventional register-flushing technique.
- We present a collective test executions' graph checking algorithm that significantly reduces the amount of computation to validate memory consistency, by 81% on average, compared to a conventional, individual execution's graph checking.
- We validate that our solution is minimally intrusive, and capable of finding subtle bugs through bug-injection case-studies conducted in a full-system simulator.
- We quantify and validate non-deterministic memory orderings in two bare-metal systems, an x86-based desktop and an ARM-based SoC platform, across various constrained-random test configurations.

2 BACKGROUND AND MOTIVATION

In multi-core systems, cores usually share caches and one or more main memory blocks, which are connected through on-chip interconnects. Since data can reside in any of the caches and memory, access latency may significantly vary among different data accesses, depending on where the actual data resides. Cache coherence protocol and memory consistency model, which support the transfer of data between cores and storage units and are responsible for enforcing retrieval and update policies, may also affect the variability of access latency, since they involve complex interactions among caches. Consequently, multi-threaded programs experience a variety of memory-access interleavings.

In this work, we focus on checking that the memory-access interleavings observed during tests executions comply with the memory consistency model (MCM). An MCM provides an interface between hardware designers and software developers, by specifying what memory re-orderings are acceptable and could be observed during a software execution [2, 3, 29, 42, 44]. For instance, with reference to the simple program in Figure 2, there are 4 different outcomes allowed by a weak MCM (*e.g.*, relaxed memory order (RMO) [9, 37, 47]): *r0* with either 0 or 1, and *r1* with either 0 or 1. Some of these outcomes are forbidden in a stronger MCM (*e.g.*, total store order (TSO) [25, 42]). As pro-

gram's size increases, so does the number of memory accesses and of re-orderings.

Constraint graph. To validate the correct implementation of the MCM in post-silicon, or any other simulation-based environment, the observed ordering of memory operations must be recorded on the fly. These access logs are then checked against the re-ordering rules allowed by the MCM of the multi-core architecture. Several works have proposed to use a constraint graph to validate the observed access sequence (*e.g.*, [4, 13, 14, 16, 24, 32, 33, 35, 36]). The vertices of this graph correspond to each memory operation (either store or load), while edges mark a dependency between two operations. Figure 2 provides the constraint graph for a simple two-threaded program execution under the TSO model. The outcome shown in the figure reveals a consistency violation because there is a cyclic dependency in the corresponding constraint graph. Indeed, in TSO, store operations cannot be speculatively performed before a preceding load operation, thus *r0* and *r1* cannot be both equal to 1 at the end of the execution. The violation is thus indicative of a microarchitectural optimization that allows multiple outstanding memory operations, against the model's requirements.

Constraint graph construction has also been thoroughly studied in recent years. In this work, we adopt the same notation as in previous work [4, 32] and model three types of 'observed' edges as follows: *reads-from* (rf), *from-read* (fr), and *write serialization* (ws). Besides observed edges, we also model intra-thread consistency edges as defined by the MCM. For instance, in TSO, the only reordering allowed is that loads can be completed before the preceding store operation.

Note that checking for cyclic dependencies in a constraint graph is usually a computation-heavy task. There are two conventional approaches: topological sorting and depth-first search (DFS). For both methods, the computational complexity is known to be $\Theta(V+E)$ where V and E are the set of vertices and edges, respectively [17]. In post-silicon validation, where many test results are generated at the system's native speed, checking these graphs can be the key bottleneck of the entire memory validation process, as we show in Section 6.2.

To **capture memory-access interleavings** at runtime, test programs must be instrumented as in prior works [24, 35]. Specifically, every store operation is assigned a unique ID, which is the value actually written into memory, so that the operation can be easily identified by subsequent loads. In addition, memory-access interleavings are captured by inspecting the values read by load operations. Thus, we can establish the uniqueness of a test execution based on its *reads-from* relationships; two executions have experienced distinct memory access interleavings when they exhibit at least one different *reads-from* relationship.

Testing framework. In this work, we adopt a constrained-random testing framework as follows: we first generate a number of test programs that are designed to likely exhibit numerous distinct memory-access interleavings. Each test program is run repeatedly until rare interleavings are observed. There are a number of prior studies focusing on test generation (Section 9); in contrast, we focus mainly on the subsequent verification steps.

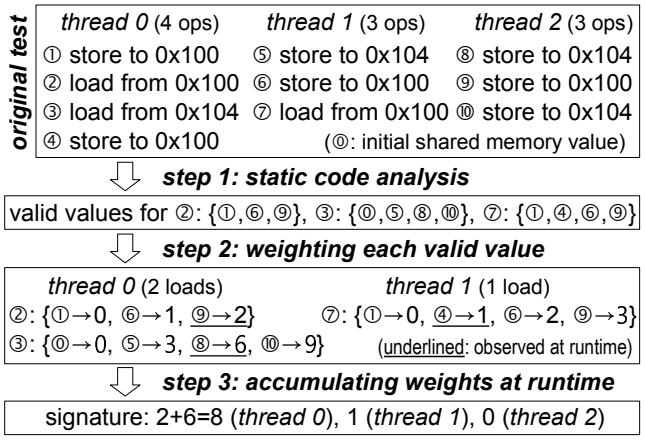

Figure 3: Memory-access interleaving signature. Each signature value corresponds to a unique memory-access interleaving, observed during the test run. For each load, we profile all possible values, and each is assigned an integer weight. At runtime, the weights of the observed values are accumulated, forming a per-thread signature. Per-thread signatures are then gathered to form an execution signature.

Specifically, we strive to (1) improve the observability of the memory-access interleavings occurring during test execution while minimally perturbing the original access sequences. We also want to (2) reduce the computation requirements for violation checking, so to boost the coverage and efficiency that can be attained in validating the MCMs in post-silicon [27].

3 CODE INSTRUMENTATION

This section discusses our code instrumentation approach that records the memory-access interleavings observed during a test's execution in post-silicon validation. Our solution is inspired by the control-flow profiling methodology proposed in [11]. However, instead of profiling control-flow paths, we repurpose the profiling framework to log memory-access interleavings in a compact signature format, that will then be categorized and checked during the subsequent checking step, discussed in Section 4.

3.1 Memory-access interleaving signature

To track memory-access interleavings, we need to log the values that are being loaded throughout the test execution. This task can be carried out by simply flushing the values loaded into the register file to a dedicated memory area, as illustrated in [24]. Lacking a dedicated storage resource and transfer channel, this task is bound to interfere with the execution flow of a test, possibly altering the latency and interleavings experienced by the test's accesses. Moreover, it burdens overall execution time with additional memory store operations.

Thus, to reduce the impact of data logging, we introduce the new concept of *memory-access interleaving signature*: MTraceCheck bypasses the frequent storing of loaded values by computing a compact signature of the loaded values. To this end, we

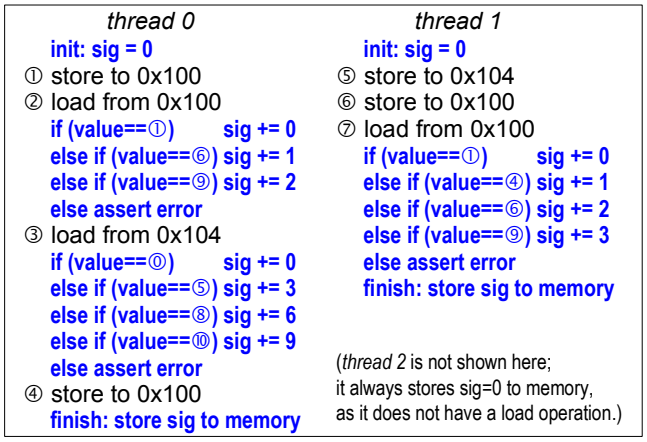

Figure 4: Code instrumentation. A signature is computed as the test runs, using chains of branch and arithmetic instructions. The computed signature is then stored in a non-shared memory region at the end of the test.

must augment the original test with signature-computation code. Note that our signature computation resembles the path-encoding computation in [11].

Figure 3 illustrates our code instrumentation process. In the first step, we perform a static code analysis to collect all possible values that could be the result of each load operation, as illustrated in the first and second boxes. This static analysis can attain perfect memory disambiguation if the test generator is configured to do so. This is straightforward to attain with a constrained-random test generator. In the case of general software test programs, it is still possible to do this analysis either (1) by excluding from the signature computation all the memory addresses that cannot be statically disambiguated, or (2) by dynamic profiling of the actual accesses using a binary instrumentation tool.

We then assign a weight to each loaded value, as illustrated in the third box of the figure. The weights are integer values consciously assigned to obtain unique final signature values. Specifically, we use consecutive integers for the first load operation. Then, if the first load operation could retrieve n distinct values, we use multiples of n for the weights of the second load operation, and so on. For instance, load operation ② can only read the value stored by either ①, ⑥ or ⑨. The weights for these values are assigned to 0, 1 and 2, respectively. Moving to the next load operation, ③, we use multiples of 3 for the weight of each possible loaded value because we had three options in the prior load operation. If there were another load operation after ④, we would use multiples of 12 for its weight, because there are already 3×4 combinations for the previous loads. By allocating non-aliasing weights (except for 0) to each loaded value and operation, we can guarantee a unique correspondence between signatures and set of loaded values, that is, a 1:1 mapping between signatures and interleavings. In the figure, suppose that two underlined values (⑨ and ⑧) are observed in thread 0, then the signature value for this thread would be 8. Note that weights

Algorithm 1 Signature decoding procedure

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

are assigned and accumulated independently for each thread. Finally, we form the *execution signature* for the test execution by concatenating the per-thread signatures obtained.

Figure 4 illustrates an example of instrumented code corresponding to the original code shown in Figure 3. The `sig` variable is initialized at the beginning of the test, and increased after each load operation. Specifically, each load operation is followed by a chain of branch statements depending on the loaded value. While not essential for observability enhancement, we append an additional assertion at the tail of the chain so that obvious errors (*e.g.*, a program-order violation) can be caught instantly without running a constraint-graph checking. This instrumentation does not perturb the sequence of memory accesses in the test execution, unless an error is caught by the assertion. With branch predictors in place, MTraceCheck only slightly increases test execution time as shown in Section 6.2. Note that this approach may entail minimal false-negatives due to these added branch operations, which could alter the original branch-prediction pattern of the test.

3.2 Per-thread signature size and decoding

In a weak MCM with no memory barrier, the signature size is proportional to the number of memory-access interleavings that are allowed by the MCM. Namely, the more diversified the interleavings possible, the larger the signature. This relationship does not hold for stronger MCMs, such as TSO, as we discuss later in Section 8.

We estimate the size of the signatures generated under the assumption that all accesses can be disambiguated and used for signature computation (as it is the case for constrained-random tests). Let T be the number of threads, S and L the number of stores and loads per thread, respectively, and A the number of shared memory locations. If assuming that addresses are uniformly randomly chosen, each load operation could read the same expected number of distinct values. The value read during a load is either the last stored value from the same thread (the 1 in the expression), or any of the stored values ($\frac{S}{A}$) from any of the other threads ($T-1$). Hence, we estimate that each per-thread signature must be capable of representing

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

distinct values. Note that the L power is required to extend our estimation to all the load operations in the thread. The execution

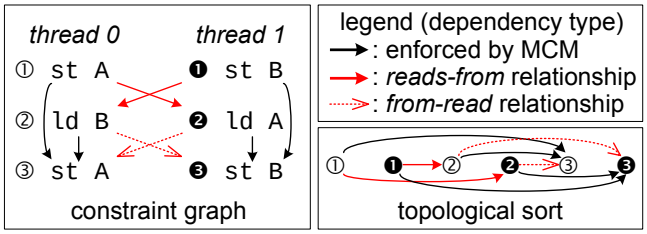

Figure 5: Topologically-sorted constraint graph. The constraint graph on the left can be topologically sorted as shown on the right, indicating no consistency violation.

signature requires T times that space, since each thread generates a per-thread signature. As an example, with S=L=50, A=32 and T=2, a per-thread signature must be capable of representing $\{1+\frac{50}{32}(2-1)\}^{50} \approx 2.7\times10^{20} \approx 2^{68}$ sets of loaded values. Thus, 68-bit of storage are required for each thread. In the systems that we used in our evaluation (Section 5), registers are either 64-bit or 32-bit wide; thus, a 68-bit signature must be split into multiple words. To support multi-word signatures, we statically detect overflow when instrumenting the tests with observability-enhancing code. When an overflow is detected, we add another register to store the signature for the thread (and another variable `sig2` in Figure 4); we then start over the signature computation in the new register, resetting the weight multipliers. In Section 6.3, we provide detailed experimental results on the average signature size for various constrained-random tests. We further discuss how to reduce the signature size in Section 8.

Reconstructing memory-access interleavings and constraint graphs. Algorithm 1 summarizes our decoding process for a per-thread signature. The goal of the reconstruction process is to translate a *signature* into a set of observed *reads-from* relationships, which will allow us to build a constraint graph for the corresponding test execution. The algorithm we developed first splits the execution signature into a set of per-thread signatures. Then it applies reconstruction to each signature obtained, starting from the last load operation in the test, and walking backwards toward the first one. Note that we maintain a special table for each test under analysis, called the *multipliers*. The table keeps track of the weight multiplier used for each load instruction in the test, and it is generated during test instrumentation. Simultaneously, we maintain the index-store mapping table for each load operation, called the *store_maps*. Thus, during reconstruction, our algorithm searches the weight multiplier to use for the load operation from the *multipliers*, and the corresponding store operation using the *store_maps* (third box in Figure 3). Then the *signature* is decreased accordingly to remove the weight component of the load just reconstructed.

Note that all other information necessary to build the constraint graph is gathered statically during the instrumentation process. That is, the *multipliers*, the *store_maps*, the intra-thread dependencies specified by the MCM (*e.g.*, load →load) and the write-serialization order. We then use this information to generate constraint graphs, as discussed in Section 2, one for each distinct test execution.

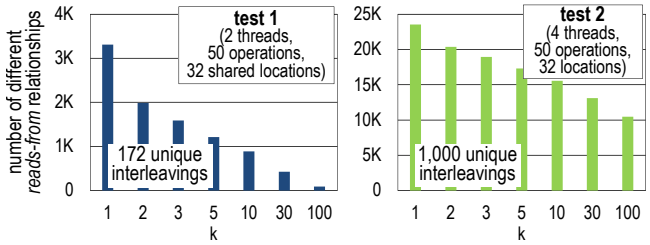

Figure 6: Measuring similarity among constraint graphs using k-medoids clustering. The number of differing *reads-from* relationships decreases as k increases. However, the cluster tightness is reduced with a more diverse pool of possible interleavings (test 2).

4 COLLECTIVE GRAPH CHECKING

As briefly introduced in Section 2, topological sorting is a classic solution to check for a memory consistency violation. *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* [17]. If a constraint graph cannot undergo topological sorting, that is an indication that a cyclic dependency is present, hence a violation of the MCM occurred. Note, however, that if some dependency edges are missing, false negatives may result from the analysis. Figure 5 shows an example of a constraint graph and its topologically sorted correspondent. In this example, there is no violation and thus a topological sort exists.

In prior works, where topological sorting was applied to constraint graphs obtained from post-silicon tests, the graph obtained from each test execution was analyzed individually. However, as we illustrate in Section 6.1, we observed that *many test runs exhibit similar memory-access interleaving patterns*. Thus, we propose to leverage the similarity among test runs to reduce the computation required to validate each execution. Below, we present a *collective* constraint-graph checking solution: we first investigate similarities among constraint graphs (Section 4.1), and then present our novel re-sorting technique that can quickly validate a collection of constraint graphs (Section 4.2). In the following discussion, we assume that all duplicate identical executions have been removed beforehand. In our implementation, duplicate executions are filtered out when execution signatures are sorted (Section 4.1).

4.1 Similarity among constraint graphs

Constraint graphs generated from a same test program have all the same vertices but possibly different edges. Thus, we identify the difference between two constraint graphs by tagging these differing edges. After we isolate the portion of a graph that differs from a previously-analyzed graph, we can isolate the discrepant portion for the sake of topological sorting. Thus, the graph-checking computation can be greatly reduced by incrementally verifying, that is, sorting only this portion.

Limit study – k-medoids clustering. While this incremental verification is promising, finding the graph most similar to the one under analysis is non-trivial and often computation-heavy. To gain insights on this aspect, we performed a preliminary study to

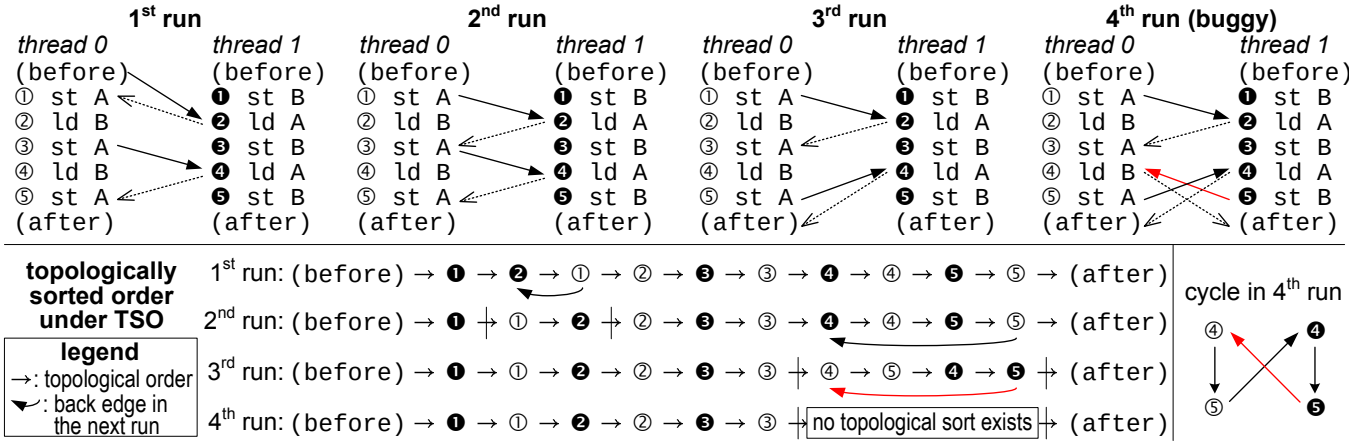

Figure 7: Re-sorting a topologically-sorted graph for a series of test runs. To check for a consistency violation, the topological sort from the previous run is partially re-sorted for the next run. The sorting boundaries (leading and trailing) are calculated from backward edges. When no topological sort exists in the segment between the boundaries, it indicates a violation.

identify a handful of graphs representative of the entire set of constraint graphs generated from many executions of a test. Specifically, we conducted a *k*-medoids clustering analysis [26] for constraint graphs from each of two test programs, measuring the total number of different *reads-from* relationships from the closest medoid graph. The selected *k* medoid graphs are considered representative of the entire set. Here, *reads-from* relationships are obtained from an in-house architectural simulator, which selects memory operations to execute in a uniformly random fashion, one at a time, and without violating sequential consistency (SC) [29]. For simplicity, we assumed single-copy store atomicity [10] for this limit study.

Figure 6 shows the number of total differing *reads-from* relationships for varying *k* values. In test 1, where we obtained 172 unique executions out of 1,000, the *reads-from* relationships decrease rapidly as *k* increases. However, in test 2, where every execution is unique, many discrepant *reads-from* relationships persist into high *k* values, indicating that the representative medoids are vastly different from the individuals. Moreover, the computational complexity of finding the optimal solution of *k*-medoids clustering is very high [26]. Thus, using this approach is computationally prohibitive, and it negates the performance benefits of a fast topological sorting.

Our solution – sorting signatures and diffing corresponding graphs. Instead of finding the graph most similar to the one under analysis, MTraceCheck opts for using a lightweight computation that finds a graph sufficiently similar to it. For this purpose, we re-use our memory-access interleaving signatures, introduced in Section 3. Once we collect all signatures from multiple executions of a test, we sort the execution signatures in ascending order. Since adjacent signatures in the order correspond to graphs with small differences between each other, we can use the graph analyzed for one signature as the basis against which to analyze the next one.

As noted in Section 3.1, *execution signatures* are generated by concatenating per-thread signature words from all test threads. Specifically, we place the signature word from the first thread in the most significant position, and the one from the last thread in the least significant position. If per-thread signatures require multiple words, we place the first signature word in the most significant position, and the last in the least significant position. To evaluate the impact of this data layout, we also performed a sensitivity study, placing signature words from related code sections in different threads near each other. This alternative layout, however, led to worse similarity between constraint graphs from adjacent signatures.

4.2 Topological order re-sorting

With the signatures sorted in ascending order, MTraceCheck examines each of the corresponding constraint graphs for consistency violations. Our checking starts with the first constraint graph in the sorted order. This first-time checking is performed as a complete, conventional graph checking, topologically sorting all vertices. Starting from the second graph, our checking algorithm strives to perform a partial re-sorting only for the portion that differs from the prior graph.

The re-sorting region is determined by two boundaries: leading boundary and trailing boundary. Only the vertices between the two boundaries need to be re-sorted. The leading (trailing) boundary is the first (last) vertex in the original sorting that is adjacent to a new backward edge from the graph under consideration. Note that neither forward nor removed edges need be considered here, since they only release prior sorting constraints. If there is no new backward edge, re-sorting is unnecessary and is thus skipped. Our re-sorting method is as precise as the conventional topological sorting; we omit the formal proof here due to limited space.

Figure 7 illustrates our re-sorting procedure on four constraint graphs obtained from a two-threaded program. The first run's topological sort is computed using a conventional complete graph checking. For the second graph, MTraceCheck examines each of the newly added edges in the graph, ①→❷ and ❷→③. Only

System 1. **x86-64** – Intel Core 2 Quad Q6600

MCM	x86-TSO [25, 42]
operating frequency	2.4 GHz
number of cores	4 (no hyper-threading support)
cache architecture	32+32 kB (L1), 8 MB (L2)
cache configuration	write back (both L1 and L2)

System 2. **ARMv7** – Samsung Exynos 5422 (big.LITTLE)

MCM	weakly-ordered memory model [7, 9]
operating frequency	800 MHz (scaled down)
number of cores	4 (Cortex-A7) + 4 (Cortex-A15)
cache architecture	A7: 32+32 kB (L1), 512 kB (L2) A15: 32+32 kB (L1), 2 MB (L2)
cache configuration	write back (L1), write through (L2)

Table 1: Specifications of the systems under validation

the former is backward, as shown by the backward arrow below the topological sort of the first run. Thus, ❷ is the leading boundary and ① is the trailing one. The order of the two nodes is then simply swapped to achieve a new topological sorting, as shown in the second run's diagram. Similarly, in the third run, four vertices must be re-sorted. The fourth run exposes a bug: indeed there is no topological sort for the four affected vertices due to the backward edge ❺→④. The absence of topological sort is also illustrated by the cycle highlighted in the bottom right part of the figure.

5 EXPERIMENTAL SETUP

Systems under validation. MTraceCheck was evaluated in two different systems, an x86-based system and an ARM-based system, as summarized in Table 1. For each system, we built a bare-metal operating environment (*i.e.*, no operating system) specialized for our validation tests. In our x86 bare-metal environment, the boot-strap processor awakens the secondary cores using inter-processor interrupt (IPI) messages, followed by cache and MMU initializations. Test threads are first allocated in the secondary cores, and then in the boot-strap core, but only when no secondary core is available. In our ARM bare-metal environment, the primary core in the Cortex-A7 cluster runs the Das U-Boot boot loader [18], which in turn, calls our test programs. At the beginning of each test, the primary core powers up the secondary cores. The secondary cores are then switched to supervisor mode and their caches and MMUs are initialized. Note that the primary core remains in hypervisor mode to keep running the boot loader.[1] Test threads are allocated in the big cores in the Cortex A15 cluster, then in the little cores in the Cortex A7 cluster.

Test generation. Table 2 shows key parameters used when generating constrained-random test programs. We chose 21 representative test configurations by combining these parameters, as shown on the x-axis of Figure 8. The naming convention for the 21 configurations is as follows: `[ISA]-[test threads]-[memory operations per thread]-[distinct shared addresses]`. For example, `ARM-2-50-32` indicates a test for the ARM ISA with 2 threads, each issuing 50 memory operations using 32 distinct shared memory addresses. For each test configuration, we generated 10 distinct tests, and ran each 5 times. The tests perform load and store instructions with equal probability (*i.e.*, load 50% and store 50%), transferring 4 bytes for each operation.

[1] We could not launch a test thread in the primary core, because of the discrepancy of the operating mode, causing unexpected hangs.

number of test threads	2, 4, 7
number of static memory operations per thread	50, 100, 200
number of distinct shared memory addresses	32, 64, 128

Table 2: Test generation parameters

Each test run executes a loop that encloses the generated memory operations so as to observe various memory-access interleaving patterns. Unless otherwise noted, the iteration count for this loop is set to 65,536. In addition, the beginning of the loop includes a synchronization routine waiting until the previous iteration is completed, followed by a shared-memory initialization and a memory barrier instruction (`mfence` for x86 and `dmb` for ARM). The synchronization routine is implemented with a conventional sense-reversal centralized barrier. To remove dependencies across test runs, we applied a hard reset before starting each test run.

6 EXPERIMENTAL RESULTS

6.1 Non-determinism in memory ordering

The dark-blue bars in Figure 8 present the number of unique memory-access interleaving patterns across various test configurations, measured by counting unique signatures. Each multi-threaded test program runs 65,536 times in a loop, except for `ARM-2-200-32*` where each test program runs 1 million iterations. We then average our findings over 5 repetitions of the same experiment. To summarize the figure, we observe almost no duplicates in several configurations on the left linear-scale graph, while we observe very few distinct interleavings in several test configurations on the right logarithmic-scale graph. For instance, `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 parameters of Table 2, the number of threads affects non-determinism the most. We measured 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 is different in these three configurations. To keep the total number of operations constant, we compared configurations with the same total number of operations (`ARM-2-100-64` versus `ARM-4-50-64`), observing again a significant increase (123 versus 22,124).

We also noted that the number of memory operations per thread affects non-determinism, but less significantly than the number of threads does. 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`. We also found that increasing the number of shared memory locations leads to fewer accesses to a same 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 of `ARM-2-200-32`.

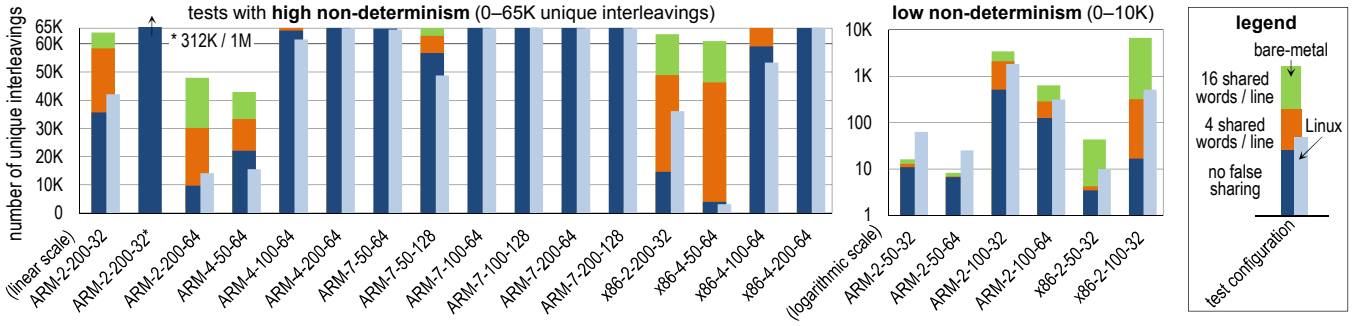

Figure 8: Number of unique memory-access interleavings. Unique memory-access interleavings diversify as more threads share the same memory space. Short two-threaded tests only exhibit a handful of distinct interleavings, while long seven-threaded tests generate new unique interleavings in almost all iterations. False sharing increases contentions, thus further diversifying interleavings. The OS contributes to creating additional unique interleavings in two-threaded tests, while the opposite trend holds in four-threaded and seven-threaded tests.

In most seven-threaded configurations, we found that almost all iterations exhibit very different memory-access patterns. This is partly because of the relatively low iteration count, which is set to 65,536. To evaluate the impact of iteration count, we performed a limited sensitivity study for `ARM-2-200-32`, where we compared the results from two iteration counts: 35,679 unique interleavings out of 65,536 (54%) versus 311,512 unique interleavings out of 1,048,576 (30%). We expect a similar trend in the seven-threaded configurations, that is, that the fraction of unique interleavings decreases as the iteration count increases.

In the x86-based system, we observe that interleaving diversity is limited compared to the ARM-based system, because the MCM of the x86-based system is stricter. We believe that the MCM is the major contributor for this difference, among other factors, such as load store queue (LSQ) size, cache organization, interconnect, *etc*. For comparison fairness, we used the same set of generated tests (*i.e.*, same memory-access patterns) for both systems.

Impact of false sharing of cache line. In cache-coherent systems like the ones we evaluated here (Table 1), the MCM is intertwined with the underlying cache-coherence protocol. Data is shared among cores at a cache-line granularity, so placing multiple shared words in a same cache line creates additional contention among threads, further diversifying memory-access interleaving patterns. For the dark-blue bars in Figure 8, only one shared word (4 bytes) is placed in each cache line (64 bytes), thus no false sharing exists.

The orange and green bars in Figure 8 present the numbers of unique memory-access interleaving patterns for two different data layouts; 4 and 16 shared words per cache line, respectively. As expected, false sharing contributes to diversifying memory-access interleaving patterns. `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. The increase is more marked for the x86-based system than for the ARM-based system.

Impact of the Operating System. Our bare-metal operating environment allows only the test program to run when testing is in progress; there is no interference with other applications. However, when our test programs are running under the control of an operating system, some test threads can be preempted by the OS scheduler. In addition, the layout of shared memory may be shuffled by the OS.

To quantify the perturbation of an OS, we re-targeted the same set of tests to a Linux environment: Ubuntu MATE 16.04 for the ARM-based system and Ubuntu 10.04 LTS for the x86-based system. Test threads are launched via the m5threads library [12], although, after launch, synchronizations among test threads are carried out by our own synchronization primitives as in the bare-metal environment.

The light-blue bars in Figure 8 report our findings for the Linux environment with no false sharing. There are two noticeable trends. Firstly, in two-threaded tests, the number of unique interleavings increases compared to the bare-metal counterparts. In both four-threaded and seven-threaded tests, on the contrary, the opposite tendency holds. We believe that fine-grained (*i.e.*, instruction-level) interferences dominate in two-threaded tests, while coarse-grained (*i.e.*, thread-level) interferences dominate in more deeply multi-threaded setups, such as four-threaded and seven-threaded tests.

6.2 Validation performance

We measured two major component costs of validation time. We first evaluated the time spent checking for MCM violations on a powerful host machine equipped with an Intel Core i7 860 2.8 GHz and 8 GB of main memory, running Ubuntu 16.04 LTS. We also report the time spent executing tests in our ARM bare-metal system. We carried out the violation checking on a host machine, instead of the system under validation, because of the heavy computation involved.

Figure 9 compares the topological-sorting time for our collective graph checking solution, normalized against a conventional

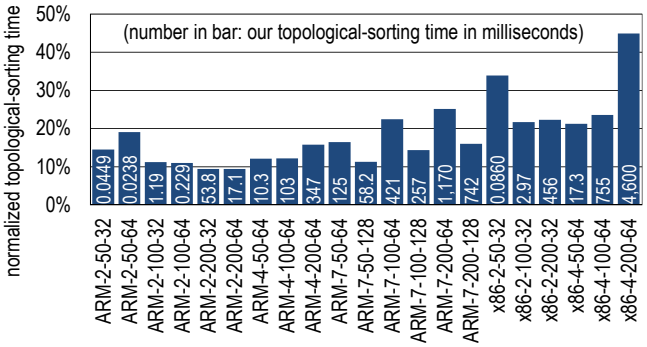

Figure 9: MCM violation checking – topological sorting speedup. MTraceCheck reduces overall topological sorting time by 81% on average, compared to a conventional individual-graph checking.

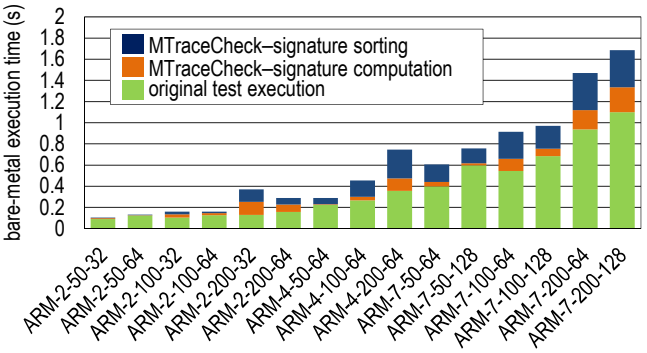

Figure 10: Test execution – MTraceCheck execution overhead. Our signature computation / sorting requires 22% / 38% of the original execution time on average, respectively.

approach that checks each constraint graph individually. We observe that our collective checking greatly reduces overall computation, consistently across all test configurations. Compared to the conventional solution, MTraceCheck takes only 9.4% (`ARM-2-200-64`) to 44.9% (`x86-4-200-64`) of the time spent by the conventional topological sorting, achieving an 81% reduction on average. We also observe a noticeable difference between the ARM and x86 platforms: the benefit of our technique is smaller in the x86 platform. We provide insights on this difference in Section 8.

In this evaluation, for reproducibility, we adopted a well-known topological-sort program, `tsort`, included with GNU core utilities [23]. We then modified the original `tsort` to support multiple constraint graphs generated from the same test. Specifically, for both the conventional and MTraceCheck techniques, vertex data structures are recycled for all constraint graphs, while edge data structures are not. The measured time excludes the time spent in reading input files, thus assuming that all graphs are loaded in memory beforehand. We ran `tsort` 5 times for each evaluation to alleviate random interferences due to the Linux environment. Due to extremely slow communication between our bare-metal systems and the host machine, we used the signatures obtained in the Linux environment (the light-blue bars in Figure 8). For fairness, we considered only unique constraint graphs for both the conventional and our techniques.

Figure 10 summarizes the test execution time, measured using performance monitors [9]. We measured three components of MTraceCheck's execution: (1) the execution time of the *original test*, (2) the execution time of our observability-enhancing code (*signature computation*), and (3) the time spent in sorting signatures (*signature sorting*).[2]

The *original test* takes 0.09–1.1 seconds to run 65,536 iterations. Our *signature computation* minimally increases the execution time: from as low as 1.5% (`ARM-2-50-64`) to up to 97.8% in an exceptional worst case (`ARM-2-200-32`). The lowest increase can be explained by noting that the test generated only about 7 unique interleaving patterns, thus the branch predictor can almost perfectly predict branch directions for our instrumented code. On the contrary, `ARM-2-200-32` exhibits a wide variety of distinct interleaving patterns, thus the performance penalty of the branch mis-predictions becomes noticeable. Our *signature sorting* algorithm also contributes to increasing the execution time, ranging from 3.9% (`ARM-2-50-64`) to 93.5% (`ARM-2-200-32`). While not shown in the graph, we observed a 140% signature-sorting overhead in the 1 million-iteration version of `ARM-2-200- 32`. Note that the signature computation and sorting overheads can be high when there are a small number of threads in the test, because the execution of the original test entails only a minimal amount of cache coherence messages.

[2]We implemented the signature-sorting program by using a balanced binary tree, written in C. We ran this program on the primary core in the Cortex-A7 cluster, after test executions completed.

6.3 Intrusiveness

To quantify the intrusiveness of MTraceCheck, we measured the amount of memory accesses unrelated to the original test execution, as shown in Figure 11. Compared to previous work [24], where all loaded values were stored back to memory, our signature-based technique requires only 7% additional memory accesses on average, ranging from 3.9% (`ARM-2-100-64`) to 11.5% (`ARM-7-200-64`). The variation can be explained by noting that test configurations providing higher data contention, *i.e.*, tests with more threads, more memory operations, and fewer shared locations, lead to a bigger per-thread signature footprint and in turn, to more data transferred due to signature collection.

Inside each bar of the figure, we report the average size of an execution signature. In low-contention configurations, the signature size is bounded by the register bit width. For instance, we need 16 bytes in `x86-2-50-32`. Note that in this configuration, the per-thread signature size merely exceeds 32 bits per thread, which leads to an average execution signature size of 8.4 bytes in `ARM-2-50-32`. However, the instrumented code uses the entire 64 bits of a register, even when fewer are needed. In high-contention configurations, the gap between 32-bit and 64-bit registers becomes narrow, as the per-thread signature requires multiple words. An extreme case is illustrated by `ARM-7-200-64`, where the signature size is 324 bytes on average (46 bytes per thread).

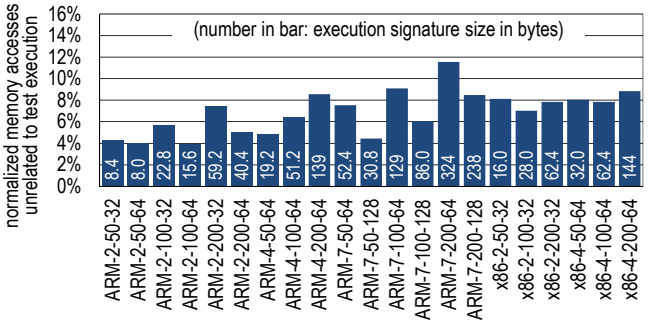

Figure 11: Intrusiveness of verification. Memory accesses unrelated to test execution are minimal compared to a register-flushing approach, only 7% on average.

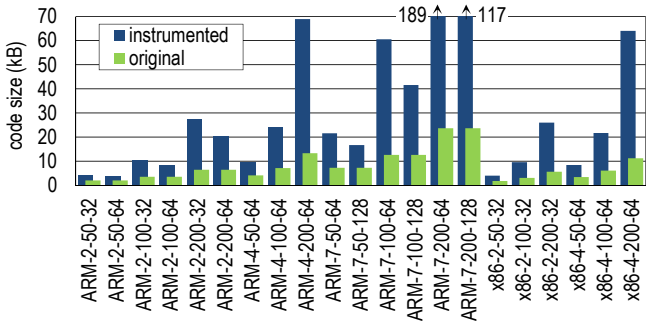

Figure 12: Code size comparison. Instrumented code is 3.7 times larger than its original counterpart, on average. However, all instrumented tests are still small enough to fit in the L1 caches.

A drawback of our solution is its increased code size. Figure 12 quantifies this aspect across various test configurations. We measured only the size of test routines, excluding initialization and signature sorting sections. The ratio of the size of the instrumented test to the original test ranges from 1.95 (`ARM-2-50-64`) to 8.16 (`ARM-7-200-64`). While this increase is conspicuous, the code size is still small enough to fit in the system's L1 instruction caches for all test configurations. For example, in `ARM-7-200-64`, the instrumented code is 189 kB; when divided by the number of threads, each core's code becomes 27 kB, fitting well in the 32 kB L1 instruction cache. Thus, the increased code size merely impacts locality aspects in instruction caches. A similar observation is drawn from Figure 10, where signature computation marginally increases the execution time.

7 BUG-INJECTION CASE-STUDIES

We performed bug-injection experiments using the gem5 simulator [12]. Three real bugs, which have been recently reported in previous work [19, 28, 32], were chosen for injection. We confirm that these bugs had been fixed in the recent version of gem5 (tag `stable_2015_09_03`). To recreate each of the bugs, we searched the relevant revision from the gem5 repository [22] and reverted the change.

We configured gem5 for eight out-of-order x86 cores with a 4×2 mesh network and a MESI cache coherence protocol with directories 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) so as to intensify the effect of cache evictions under our small working set, while the rest of the configuration is modeled after a recent version of the Intel Core i7 processor. We compiled our tests with the m5threads library to run simulations under gem5's syscall emulation mode.

7.1 Bug descriptions

Bug 1 – load→load violation 1 (protocol issue): Bug 1 (fixed in June 2015) is modeled after "MESI,LQ+SM,Inv" described in [19]. This bug is a variant of the Peekaboo problem [44], where a load operation appears to be performed earlier than its preceding load operation. It is triggered when a cache receives an invalidation for a cache line that is currently transitioning from shared state to modified state. When the cache receives the invalidation message, subsequent load operations to this cache line must be re-tried after handling the received invalidation. However, this bug prevents the cache from squashing the subsequent load operations, causing a load→load violation.

Bug 2 – load→load violation 2 (LSQ issue): Bug 2 (fixed in March 2014) is reported by two independent works: [32] and [19]. This bug is similar to bug 1 in its manifestation, but caused by an LSQ that does not invalidate subsequent loads upon the reception of an invalidation message.

Bug 3 – race in cache coherence protocol: Bug 3 (fixed in January 2011) is modeled after the "MESI bug 1" detailed in [28], which is also evaluated in [19]. This bug is triggered by a race between an L1 writeback message (PUTX) and a write-request message (GETX) from another L1.

7.2 Bug-detection results

For each bug, we deliberately chose the test configuration in Table 3 after analyzing the bug descriptions,[3] and generated 101 random tests. The iteration count was greatly reduced to 1,024 from 65,536 in Section 6, because of the much slower speed of gem5 simulations, compared to native executions.

We report our bug-detection results on the third column of Table 3. Overall, MTraceCheck successfully found all injected bugs. Note that bug 1 was detected by only one test (out of 101): in this test, we found that 29 signatures (out of 1,024 unique signatures) exhibited invalid memory-access interleavings. Bug 2 was easier to expose than bug 1; ten tests revealed 1 invalid signature each, with one test revealing 2. The impact of bug 3 is much more dramatic, crashing all gem5 simulations with internal error messages (*e.g.*, protocol deadlock and invalid transition).

bug	test configuration	bug detection results
1	`x86-4-50-8` (4 words/line)	1 test, 29 signatures
2	`x86-7-200-32` (16 words/line)	11 tests, 12 signatures
3	`x86-7-200-64` (4 words/line)	all tests (crash)

Table 3: Bug detection results

[3] We were able to find bugs 2 and 3 without much effort in selecting test configurations. For bug 1, we also tried the same seven-threaded random tests, as well as a few hand-crafted tests, to no avail. With advanced test-generation techniques (Section 9), the bug detection rate can be greatly improved.

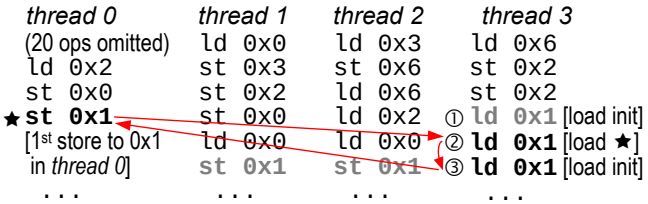

Figure 13: Detected load→load ordering violation. Loads ② and ③ show an invalid re-ordering due to bug 1.

Figure 13 provides a snippet of the test exposing bug 1. The first 20 memory operations of thread 0 are omitted for simplicity. The starred instruction ★ is the first store to memory address `0x1`. Both threads 1 and 2 perform a store operation to `0x1`, while thread 3 executes three consecutive load operations from `0x1`. Among these three loads, the first and the third read the initial value of the memory location, while the second reads the value from ★. From the second and third loaded values, we identify a cyclic dependency illustrated with red arrows in the figure: ★ happens before ② (*reads-from*), which should happen before ③ (load→load ordering), which in turn happens before ★ (*from-read*).

8 INSIGHTS AND DISCUSSIONS

We attribute the speedup of our collective graph checking (Figure 9) to two factors. Firstly, we notice that many constraint graphs can be validated immediately without modifying the previous topological sort. This situation is highlighted in the leftmost bar (ARM) of Figure 14, where all constraint graphs beside the first do not require any re-sorting. `tsort` unwittingly places store operations prior to load operations since stores do not depend on any load operations in absence of memory barriers, as it is the case for our generated tests. Note that `tsort` is MCM-agnostic, so it still needs to check every new backward edge.

Secondly, for x86 tests, we also observe that a majority of constraint graphs are incrementally checked. The blue bars in Figure 14 show the percentage of such graphs, ranging from 82% (`x86-2-50-32`) to almost 100% (`x86-4-200-64`). For these graphs, we measure the average percentage of vertices affected by re-sorting in the line plotted in the figure: this percentage ranges from 21% (`x86-4-50-64`) to 78% (`x86-4-200-64`). It is the high incidence of re-sorting that negatively affected the violation checking performance boost shown in Figure 9.

Pruning invalid memory-access interleavings. Our code instrumentation (Section 3) makes a conservative assumption to support a wide range of MCMs in a single framework; each memory operation can be independently reordered, regardless of its preceding operations. This conservative assumption comes at the cost of increased signature and code sizes (Figures 11 and 12). To address these two drawbacks, we can leverage microarchitectural information when instrumenting observability-enhancing code (*static pruning*). For instance, the number of outstanding load and store operations are often bounded by the number of LSQ entries, *etc*. Using this additional information, we can reduce the number of options for loaded values (as in [43]), thus decreasing the sizes

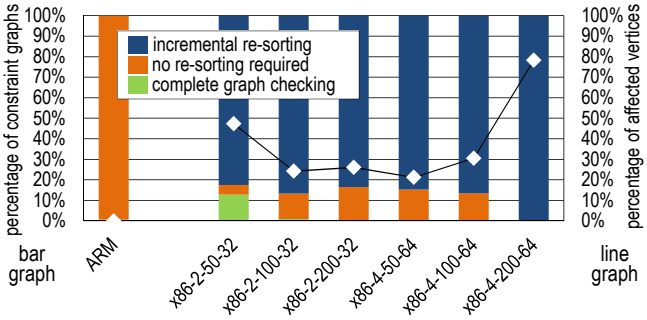

Figure 14: Breakdown of our collective graph checking. For ARM tests, most of graphs do not require any re-sorting. For x86 tests, up to 16% of the graphs can bypass re-sorting. Among the others, up to 78% of the vertices are affected.

of instrumented code and signatures. In our real-system evaluations, however, we could not gather sufficient microarchitectural details to employ this optimization.

Moreover, in a strong MCM (*e.g.*, TSO), we can also apply a runtime technique to reduce signature size (*dynamic pruning*). At runtime, each thread would track the set of the recent store operations performed by other threads, computing a frontier of memory operations. Any value loaded from a store operation behind this frontier would be considered invalid. However, with this dynamic pruning, signature decoding becomes complicated as the length of signatures varies. Also, instrumented code size further increases due to frontier computation.

Scalability. As discussed in Section 3.2, the size of a per-thread signature increases quickly as the numbers of threads and memory operations increase. Although we demonstrated MTraceCheck with test-cases that are much larger than typical litmus tests, even larger test-cases can be obtained by merging multiple independent code segments, where memory addresses are assigned in a way that leads only to false sharing across the segments.

Store atomicity. Our proposed solution (Sections 3 and 4) makes no assumption on store atomicity (single-copy atomic, multiple-copy atomic, or non-multiple-copy atomic). However, our evaluation (Sections 5 through 7) does not include a single-copy atomic system.[4] We expect that constraint graphs for such systems would need additional dependency edges [10, 33]. We also expect that our collective checking would still perform better than conventional checking, but the advantage would decrease due to larger re-sorting windows.

9 RELATED WORK

Dynamic validation of memory consistency. With the growing popularity of multi-core systems, validating memory consistency has gained research interest in both academia and industry. One class of research work takes a dynamic validation approach, striving to check the memory re-ordering observed while running

[4]While developing MTraceCheck, we encountered false positives by wrongly assuming single-copy atomicity on the x86-based system. These false positives have been resolved by ignoring intra-thread store-load dependency edges [10].

either a constrained-random test or a real application. TSOtool [24] is one of the early efforts to validate SPARC TSO [47] using a constraint graph approach. MTraceCheck improves on this work by reducing the computation requirements of constraint-graph checking and by minimizing memory-access perturbation. DVMC [38] proposes to augment the memory subsystem to verify memory re-orderings at runtime. Chen *et al*. [14] take a similar approach, where each memory operation is piggybacked with a unique ID. Compared to these two works, MTraceCheck is a purely software-based approach that does not require any hardware support. Also, we strive to find bugs in the post-silicon validation stage, while those two works are specialized for runtime validation, after deployment. LCHECK [16] provides a fast validation framework for systems with store atomicity [10]. The relaxed scoreboard solution by [43] tracks a tight set of valid loaded values for pre-silicon validation. DACOTA [35] proposes a hardware-software hybrid solution for post-silicon MCM validation.

Formally verifying MCMs. Another class of research work takes a formal verification approach, where the memory consistency is modeled as a mathematical, abstract object. The abstract model captures the dependencies among instructions and micro-architectural components. The model is then verified by checking whether it satisfies desired properties in the MCM. Alglave [4] verifies several weak MCMs using the Coq theorem prover [45]. PipeCheck [32] improves the verification accuracy by leveraging additional micro-architectural information, proposing μhb graphs, which can be also used to verify the interface between cache coherence and memory consistency [36]. Note that these proposals use relatively small litmus tests, because the theorem prover must enumerate every possible memory-access interleaving for the given tests. With increasing test sizes, however, theorem proving scales poorly, limiting its application to post-silicon tests. Fractal consistency [48] proposes to design MCMs in a scalable manner, addressing the scalability challenge of verification. ArMOR [33] aims to accurately specify MCMs for a dynamic binary translation between heterogeneous ISAs.

Test generation for memory consistency validation. Various litmus tests (*e.g.*, [5–7, 34]) have been developed and are widely used in memory consistency validation. When validating full systems, however, these litmus tests are often insufficient, as illustrated in [19]. Constrained-random test generators (*e.g.*, [24, 40, 41]) strive to validate corner cases uncovered by the litmus tests. McVerSi [19] presents a test-generation framework using genetic programming, improving the test coverage over a random generation. In our evaluation, we only used a constrained-random generator; however, advanced test generation can be paired with our code instrumentation and checking techniques. Program regularization [15] proposes to augment test programs with additional inter-thread synchronizations to simplify the frontier computation (as discussed in Section 8).

Post-silicon microprocessor validation aims at finding various hard-to-find errors in silicon chips. Constrained-random tests are heavily used in this validation stage [1, 24], often paired with self-checking solutions [20, 21, 31, 46]. These solutions strive to overcome the observability challenge in post-silicon validation [27, 39], by checking test results directly on chip, without transferring results to a host machine. However, the solutions proposed so far rely on deterministic behaviors of program executions. Moreover, many of these solutions restrict test generation to a great extent. For example, Reversi [46] relies on reversible instructions (*e.g.*, a pair of `add` and `sub`). As we reported in Section 6.1, memory-access interleavings are fundamentally diverse, and thus these self-checking solutions become ineffective. Our proposal strives to efficiently validate non-deterministic, diverse behaviors.

10 CONCLUSIONS

Verifying non-deterministic behaviors is an important and demanding task in post-silicon microprocessor validation. Our proposal, MTraceCheck, strives to efficiently validate a wide range of non-deterministic behaviors in memory-access interleavings observed in multi-core systems. We tackle two major obstacles in accomplishing this goal in post-silicon validation: limited observability and heavy result-checking computation. To alleviate the first obstacle, we present a signature computation method that encapsulates observed memory-access interleavings in a compact signature. This observability enhancement minimally perturbs the memory accesses of the original test, thus it is capable of exposing subtle hardware bugs. To mitigate the second obstacle, our collective graph checking algorithm reduces computation requirements by leveraging the structural similarities among constraint graphs. We have applied MTraceCheck to two silicon systems with various test configurations, consistently achieving a remarkable speedup in validating non-deterministic behaviors.

Acknowledgement. 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] Allon Adir, Maxim Golubev, Shimon Landa, Amir Nahir, Gil Shurek, Vitali Sokhin, and Avi Ziv. 2011. Threadmill: A Post-Silicon Exerciser for Multi-Threaded Processors. In *Proceedings of the 48th Design Automation Conference*. 860–865. https://doi.org/10.1145/2024724.2024916

[2] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (1996), 66–76. https://doi.org/10.1109/2.546611

[3] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering–A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. 2–14. https://doi.org/10.1145/325164.325100

[4] Jade Alglave. 2012. A Formal Hierarchy of Weak Memory Models. *Formal Methods in System Design* 41, 2 (2012), 178–210. https://doi.org/10.1007/s10703-012-0161-5

[5] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification: 22nd International Conference, CAV 2010*. 258–272. https://doi.org/10.1007/978-3-642-14295-6_25

[6] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware. In *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011*. 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

[7] ARM. 2009. *Barrier Litmus Tests and Cookbook*.

[8] ARM. 2011. *Embedded Trace Macrocell Architecture Specification*.

[9] ARM. 2012. *ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition*.

[10] Arvind and Jan-Willem Maessen. 2006. Memory Model = Instruction Reordering + Store Atomicity. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. 29–40. https://doi.org/10.1109/ISCA.2006.26

[11] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. 46–57. https://doi.org/10.1109/MICRO.1996.566449

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718

[13] Harold W. Cain, Mikko H. Lipasti, and Ravi Nair. 2003. Constraint Graph Analysis of Multithreaded Programs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. 4–14. https://doi.org/10.1109/PACT.2003.1237997

[14] Kaiyu Chen, Sharad Malik, and Priyadarsan Patra. 2008. Runtime Validation of Memory Ordering Using Constraint Graph Checking. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 415–426. https://doi.org/10.1109/HPCA.2008.4658657

[15] Yunji Chen, Lei Li, Tianshi Chen, Ling Li, Lei Wang, Xiaoxue Feng, and Weiwu Hu. 2012. Program Regularization in Memory Consistency Verification. *IEEE Transactions on Parallel and Distributed Systems* 23, 11 (2012), 2163–2174. https://doi.org/10.1109/TPDS.2012.44

[16] Yunji Chen, Yi Lv, Weiwu Hu, Tianshi Chen, Haihua Shen, Pengyu Wang, and Hong Pan. 2009. Fast Complete Memory Consistency Verification. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 381–392. https://doi.org/10.1109/HPCA.2009.4798276

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[18] Das U-Boot – the universal boot loader. Retrieved April 30, 2017 from http://www.denx.de/wiki/U-Boot

[19] Marco Elver and Vijay Nagarajan. 2016. McVerSi: A Test Generation Framework for Fast Memory Consistency Verification in Simulation. In *2016 IEEE International Symposium on High Performance Computer Architecture*. 618–630. https://doi.org/10.1109/HPCA.2016.7446099

[20] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. 2011. Accelerating Microprocessor Silicon Validation by Exposing ISA Diversity. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 386–397. https://doi.org/10.1145/2155620.2155666

[21] Nikos Foutris, Dimitris Gizopoulos, Xavier Vera, and Antonio Gonzalez. 2013. Deconfigurable Microprocessor Architectures for Silicon Debug Acceleration. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 631–642. https://doi.org/10.1145/2485922.2485976

[22] gem5 mercurial repository host. Retrieved April 30, 2017 from http://repo.gem5.org

[23] GNU coreutils version 8.25. Retrieved April 30, 2017 from http://ftp.gnu.org/gnu/coreutils/

[24] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, Juin-Yeu J. Lu, and Sridhar Narayanan. 2004. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. 114–123. https://doi.org/10.1109/ISCA.2004.1310768

[25] Intel. 2015. *Intel 64 and IA-32 Architectures Software Developer's Manual*.

[26] k-medoids algorithm. Retrieved April 30, 2017 from https://en.wikipedia.org/wiki/K-medoids

[27] Jagannath Keshava, Nagib Hakim, and Chinna Prudvi. 2010. Post-Silicon Validation Challenges: How EDA and Academia Can Help. In *Proceedings of the 47th Design Automation Conference*. 3–7. https://doi.org/10.1145/1837274.1837278

[28] Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. 2014. Revisiting the Complexity of Hardware Cache Coherence and Some Implications. *ACM Transactions on Architecture Code Optimization* 11, 4 (2014), 37:1–37:22. https://doi.org/10.1145/2663345

[29] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[30] Penny Li, Jinuk L. Shin, Georgios Konstadinidis, Francis Schumacher, Venkat Krishnaswamy, Hoyeol Cho, Sudesna Dash, Robert Masleid, Chaoyang Zheng, Yuanjung D. Lin, Paul Loewenstein, Heechoul Park, Vijay Srinivasan, Dawei Huang, Changku Hwang, Wenjay Hsu, and Curtis McAllister. 2015. A 20nm 32-Core 64MB L3 Cache SPARC M7 Processor. In *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*. 1–3. https://doi.org/10.1109/ISSCC.2015.7062931

[31] David Lin, Ted Hong, Yanjing Li, Eswaran S, Sharad Kumar, Farzan Fallah, Nagib Hakim, Donald S. Gardner, and Subhasish Mitra. 2014. Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10 (2014), 1573–1590. https://doi.org/10.1109/TCAD.2014.2334301

[32] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2014. PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 635–646. https://doi.org/10.1109/MICRO.2014.38

[33] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 388–400. https://doi.org/10.1145/2749469.2750378

[34] Sela Mador-Haim, Rajeev Alur, and Milo M.K. Martin. 2010. Generating Litmus Tests for Contrasting Memory Consistency Models. In *Computer Aided Verification: 22nd International Conference, CAV 2010*. 273–287. https://doi.org/10.1007/978-3-642-14295-6_26

[35] Biruk W. Mammo, Valeria Bertacco, Andrew DeOrio, and Ilya Wagner. 2015. Post-Silicon Validation of Multiprocessor Memory Consistency. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 6 (2015), 1027–1037. https://doi.org/10.1109/TCAD.2015.2402171

[36] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface. In *Proceedings of the 48th International Symposium on Microarchitecture*. 26–37. https://doi.org/10.1145/2830772.2830782

[37] Luc Maranget, Susmit Sarkar, and Peter Sewell. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. Retrieved April 30, 2017 from http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf

[38] Albert Meixner and Daniel J. Sorin. 2009. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. *IEEE Transactions on Dependable and Secure Computing* 6, 1 (2009), 18–31. https://doi.org/10.1109/TDSC.2007.70243

[39] Subhasish Mitra, Sanjit A. Seshia, and Nicola Nicolici. 2010. Post-Silicon Validation Opportunities, Challenges and Recent Advances. In *Proceedings of the 47th Design Automation Conference*. 12–17. https://doi.org/10.1145/1837274.1837280

[40] Eberle A. Rambo, Olav P. Henschel, and Luiz C. V. dos Santos. 2011. Automatic Generation of Memory Consistency Tests for Chip Multiprocessing. In *18th IEEE International Conference on Electronics, Circuits, and Systems*. 542–545. https://doi.org/10.1109/ICECS.2011.6122332

[41] Amitabha Roy, Stephan Zeisset, Charles J. Fleckenstein, and John C. Huang. 2006. Fast and Generalized Polynomial Time Memory Consistency Verification. In *Computer Aided Verification: 18th International Conference, CAV 2006*. 503–516. https://doi.org/10.1007/11817963_46

[42] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Z. Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443

[43] Ofer Shacham, Megan Wachs, Alex Solomatnikov, Amin Firoozshahian, Stephen Richardson, and Mark Horowitz. 2008. Verification of Chip Multiprocessor Memory Systems Using a Relaxed Scoreboard. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. 294–305. https://doi.org/10.1109/MICRO.2008.4771799

[44] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.

[45] The Coq proof assistant. Retrieved April 30, 2017 from https://coq.inria.fr

[46] Ilya Wagner and Valeria Bertacco. 2008. Reversi: Post-Silicon Validation System for Modern Microprocessors. In *IEEE International Conference on Computer Design*. 307–314. https://doi.org/10.1109/ICCD.2008.4751878

[47] David Weaver and Tom Germond. 1994. *The SPARC Architectural Manual (Version 9)*. Prentice-Hall, Inc.

[48] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. 2010. Fractal Consistency: Architecting the Memory System to Facilitate Verification. *IEEE Computer Architecture Letters* 9, 2 (2010), 61–64. https://doi.org/10.1109/L-CA.2010.18