PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing

Themis Melissaris
Princeton University

Markos Markakis
Princeton University

Kelly Shaw
Williams College

Margaret Martonosi
Princeton University

***Abstract*—Even as most of today's computer systems have turned to parallelism to improve performance, their documentation often remains informal, incomplete or even incorrect regarding their memory consistency models. This leads to programmer and designer confusion and to buggy concurrent systems. Existing tools for empirical memory consistency testing rely on large numbers of iterations of simple multi-threaded litmus tests to perform conformance testing. The current approach typically employs thread synchronization at every iteration, which imposes a significant overhead and can reduce testing performance and efficiency.**

This paper proposes new litmus test variants called *perpetual litmus tests*, which allow for consistency testing without per-iteration synchronization. Perpetual litmus tests use arithmetic sequences in store operations to reduce the required synchronization points. We present PerpLE, a software suite that includes tools for the generation, execution, and analysis of perpetual litmus tests. We introduce an algorithm for determining the outcomes of perpetual litmus tests as well as a scalable linear heuristic algorithm. Evaluating the performance, scalability and ability of our tool to find outcomes of interest on an x86 system, we observe a wider variety of outcomes than litmus7 while experiencing runtime speedups over all litmus7 synchronization modes (8.89x over the default `user` mode). Compared to the default litmus7 synchronization (`user`) mode, PerpLE offers over four orders-of-magnitude improvement in the rate with which we detect target outcomes.

I. Introduction

As traditional approaches to increasing system performance have been constrained by the end of Moore's Law/ Dennard scaling, exploiting parallelism has become the primary way to further improve performance across system types [3, 4, 23]. When working with these parallel systems, understanding their memory consistency models is critically important for correct design and programming. Otherwise, the resulting systems and applications can contain correctness bugs that manifest as subtle, non-deterministic errors [39].

As hardware complexity rises, it is becoming increasingly challenging to ensure that a specific implementation conforms to the memory model it claims to implement, generating the need for thorough testing. Time spent in validation and verification can be more than half of the hardware design effort [19, 20], which highlights the need for approaches that can accelerate these processes; faster methodologies can, in turn, reduce cost and improve time-to-market. While some comprehensive formal techniques exist for exploring litmus test execution [30, 31], industry testing of real hardware relies more on empirical and probabilistic testing [24], as it is likely to provide results at a significantly shorter time scale. Most current approaches to this type of testing leverage small parallel programs called litmus tests, designed to expose different orderings of memory operations. Orderings that manifest in the empirical execution of litmus tests are called *observable* for an implementation. Observing an ordering that the system's published memory model lists as forbidden indicates an implementation bug; to maximize the probability of detecting bugs, empirical testing tools aim to expose as large a variety of outcomes as possible.

Currently, tools achieve outcome variety by executing litmus tests iteratively, with different orderings arising probabilistically during test execution due to factors like system load and thread timing [24]. However, each litmus test might need to be run thousands or even millions of times before the desired testing outcomes are observed [39]. The frequency with which a given test outcome, indicative of a particular ordering, can be observed depends on (i) the extent to which the implementation under test favors it (including whether it is technically feasible) and (ii) the ability of the testing approach to create the conditions that would reveal it. For less frequent outcomes, some testing approaches may require executing large numbers of iterations, taking significant amounts of testing time, to observe a desired outcome. PerpLE is designed to more efficiently expose and analyze a wide range of orderings, improving the effectiveness of these empirical testing approaches.

While tools like litmus7, provided by the *diy* suite [11, 24], are effective empirical testing frameworks, many rely on synchronizing the participating threads before every test iteration. Such synchronization is critical to ensure that different threads execute their part of the test sufficiently close in time for the testing tool to observe their interaction via shared memory, but it can have negative implications. The synchronization overhead dominates runtime, significantly slowing down testing and reducing the total number of iterations executed. For example, based on our experiments on litmus7 using the default (`user`) synchronization mode and for different iteration counts of the store buffering (sb) litmus test, synchronization overhead never falls below 85% of total execution time. Furthermore, the tight synchronization might reduce the number and type of orderings that are ever experienced during the iterative test. Lastly, iterative synchronization-based testing can be ineffective for systems that incur long synchronization overheads compared to CPUs, e.g. GPUs, as well as for systems not optimized

for performance, like many mobile processors [13]. In fact, consistency testing approaches in these systems exhibit orders of magnitude lower performance compared to microprocessors [39]. Empirical testing of memory models in these systems might also be hindered by multi-core memory interference that can slow programs down by over 100x [25].

PerpLE is designed to increase the opportunities to observe different outcomes per unit time. Our approach rethinks the design of litmus tests to eliminate per-iteration synchronization, thus reducing the overall testing time. At the same time, letting threads run longer without synchronizing creates more opportunities for interesting interactions, leading to a greater variety of outcomes and, by extension, of orderings. While other synchronization-free litmus testing tools exist, PerpLE offers advantages over existing approaches. Specifically, this work offers the following contributions:

- We propose an empirical memory consistency testing approach without the requirement for per-iteration synchronization. Our approach is based on what we call *perpetual litmus tests*, which we define and analyze.
- We demonstrate a method for converting litmus tests to their perpetual counterparts and generate a suite of perpetual litmus tests.
- We provide a software suite capable of generating, running and analyzing such tests from regular litmus tests, which we call the Perpetual Litmus Engine (PerpLE).
- We present an algorithm for detecting outcomes of interest in *perpetual litmus tests*. We also present a linear heuristic that allows PerpLE to scale to millions of test iterations while maintaining a rate of outcome discovery that is orders of magnitude higher than existing approaches.
- We evaluate PerpLE on an x86 system, showing increased outcome variety and target outcome occurrence rate compared to a variety of litmus7 modes, including its synchronization-free mode.
- We find PerpLE achieves a runtime speedup over all litmus7 synchronization modes (8.89x over `user` mode and 2.52x over the synchronization-free, `none` mode).
- Finally, we also observe an increase of over four orders of magnitude over the default synchronization mode of litmus7 (`user` mode) in our chosen metric of outcomes of interest observed per unit time, which makes testing practical across parallel hardware systems.

II. Background

A. Memory Consistency Models

1) Sequential Consistency: In a multi-core context, memory consistency models define the rules that determine the ordering of concurrent loads and stores, and therefore determine which values can be legally returned to program loads. In Lamport's *sequential consistency (SC)* [28], all threads see loads and stores to memory in the same globally agreed-upon order. The concurrent execution of shared memory operations by different threads can be viewed as a single thread executing some interleaving of these operations without violating the program order of each thread, as shown in the bottom left of Figure 1.

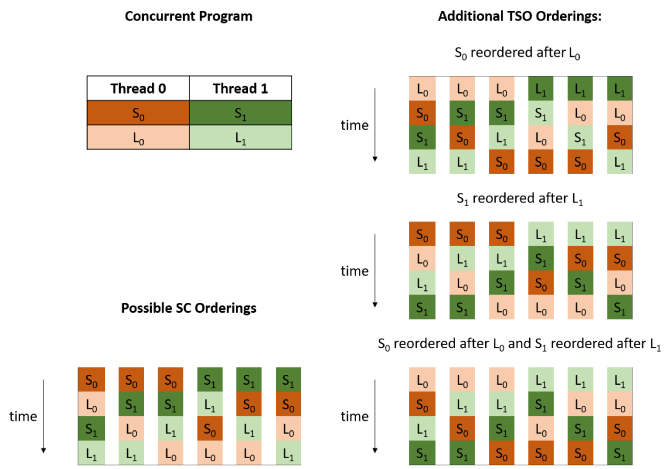

Fig. 1: Possible memory operation orderings under SC and TSO. S_0 and S_1 are stores, while L_0 and L_1 are loads. Under SC, the possible orderings are interleavings of the memory operations of each thread in program order. TSO relaxes the program order requirement by allowing store buffering, so stores can appear to take effect later, yielding additional allowed orderings.

Even though sequential consistency is very intuitive, it is rarely found in real systems, because it limits memory system concurrency [14] [29]. Modern processors usually relax the guarantee that the per-thread program order is preserved globally and turn to other weaker memory models instead.

2) TSO: Total Store Ordering (TSO) can be informally understood as the memory model that is obtained from SC by introducing store buffering to reduce the latency of store operations. Each thread can load its own stored values early, directly out of its store buffer, but all threads share the same view of the global order in which stores appear to have occurred. This can provide additional allowed orderings in some cases, as shown in the right column of Figure 1.

Owens et al. have shown that a version of TSO is consistent with the behavior of x86 processors across a number of test cases [37, 38]. As such, TSO is one of the predominant memory consistency models one encounters when analyzing the memory behavior of CPUs. Our work focuses on TSO here, but is applicable to weaker models as well.

B. Uncovering Instruction Orderings

1) Litmus tests: In order to test that software and hardware systems adhere to their specified memory models, small stressmark tests known as *litmus tests* are used. Litmus tests consist of simple combinations of shared memory loads and stores. Figure 2 shows three example litmus tests: store buffering (sb), load buffering (lb) and podwr001. For each litmus test, we use T to denote the total number of test threads and $T_L \leq T$ to denote the number of test threads that perform loads.

sb

Initially [x] = 0, [y] = 0	
Thread 0	**Thread 1**
(i_{00}): [x] ← 1	(i_{10}): [y] ← 1
(i_{01}): reg_0_0 ← [y]	(i_{11}): reg_1_0 ← [x]

lb

Initially [x] = 0, [y] = 0	
Thread 0	**Thread 1**
(i_{00}): reg_0_0 ← [y]	(i_{10}): reg_1_0 ← [x]
(i_{01}): [x] ← 1	(i_{11}): [y] ← 1

podwr001

Initially [x] = 0, [y] = 0, [z] = 0		
Thread 0	**Thread 1**	**Thread 2**
(i_{00}): [x] ← 1	(i_{10}): [y] ← 1	(i_{20}): [z] ← 1
(i_{01}): reg_0_0 ← [y]	(i_{11}): reg_1_0 ← [z]	(i_{21}): reg_1_0 ← [x]

Fig. 2: Store buffering (sb), load buffering (lb) and podwr001 litmus tests. Note: podwr001 extends sb to 3 threads. (i_{tn}) is the nth test instruction of thread t, $[x]$ and $[y]$ are shared memory locations and reg_t_r is register r of thread t. For sb and lb, $T = T_L = 2$. For podwr001, $T = T_L = 3$.

Running a litmus test can produce one of a set of possible *outcomes*, depending on the values loaded from shared memory during test execution. Each outcome consists of a number of *conditions* involving register values. For example, the sb test as presented in Figure 2 has 4 possible outcomes, each consisting of 2 conditions: $reg_0_0 = 0$ && $reg_1_0 = 0$; $reg_0_0 = 0$ && $reg_1_0 = 1$; $reg_0_0 = 1$ && $reg_1_0 = 0$; and $reg_0_0 = 1$ && $reg_1_0 = 1$. In each test run, we can select a subset of these outcomes and measure how often each of them occurred; we call this subset the *outcomes of interest*.

Note that the first of these outcomes requires the hardware to implement store buffering in order to occur, i.e. it cannot occur under SC by simply interleaving the instructions from each thread. For this reason, it is the most informative outcome in terms of hardware capabilities, letting us distinguish between different possible consistency models. Each litmus test has at least one such outcome, the *target outcome*.

The outcomes of many iterative executions of a litmus test indicate which interleavings occurred. These observed interleavings can then be checked against the model that the system claims to implement to ensure they are allowed. Since there is a non-deterministic element in individual runs of a litmus test, such approaches typically cannot guarantee that all possible interleavings have been exercised.

In practice, widely available tools for litmus testing use a combination of two approaches to address this non-determinism [9, 11]. First, they run large numbers of iterations of the litmus test to give a statistically more accurate picture of how frequently each outcome is expected to appear. For models not yet formally specified, this can aid attempts at formulating a formal description. Second, testing suites might apply further stress on the system, like frequent memory operations to addresses not used by the test, to check whether the distribution of outcomes is affected. Especially in GPUs, recent work has shown that such methods can be very effective [39].

2) Happens-Before Graphs: Each litmus test outcome offers information on the memory operation ordering that gave rise to it, which can be revealed using a *happens-before graph*. Constructing a happens-before graph starts by considering the different shared memory operations in a litmus test as vertices in a graph. Edges are then added to represent temporal relationships between individual operations, based on the outcome of a specific execution of the test. Happens-before edges are meant to represent temporal relationships and as such are transitive. Alglave [8] provides formal descriptions of four types of such edges between two memory operations m_1 and m_2, summarized informally below:

- Program order (po) edges: a po edge from m_1 to m_2 means a sequential processor executes m_1 before m_2.
- Read-from (rf) edges: an rf edge from a store m_1 to a load m_2 means that m_2 loads the value stored by m_1.
- Write serialization (ws) edges: assuming m_1 and m_2 are both stores to the same memory location x, a ws edge from m_1 to m_2 means that m_1 updates x before m_2.
- From-read (fr) edges: an fr edge from a load m_1 to a store m_2 means that m_1 loads a value stored by an instruction earlier than m_2 in ws order.

III. Test Conversion to Remove Synchronization

As Figure 3 illustrates, our proposed approach consists of two steps, each performed by a separate tool. First, the *Converter* converts an input litmus test to a format capable of exposing the same interleavings as the original test, but without requiring per-iteration thread synchronization, which we call a *perpetual litmus test*. This test is then executed by the *Harness*, which keeps the test run results in memory.

Meanwhile, the Converter also produces the *exhaustive outcome counter* for this particular litmus test and set of outcomes of interest. The exhaustive outcome counter is a function that the Harness can apply to the in-memory test results, once all iterations of the test have been executed, to determine how many times each outcome of interest occurred. Alongside the exhaustive outcome counter, the Converter produces the *heuristic outcome counter*, a function with the same inputs and outputs, but which only searches part of the results space and can be dramatically faster.

The rest of this Section presents the methodology for test conversion, while Section IV deals with generating the exhaustive and heuristic outcome counter functions.

A. Synchronization in litmus tests

Since litmus tests tend to only be a few instructions long, the execution time of a single iteration is very short. Unless we synchronize before each iteration, it is most likely that the participating threads will execute their corresponding parts of the test at different points in time, so interactions among individual memory operations executed by different threads will be unlikely. Perpetual litmus tests account for this effect.

Moreover, determining the outcome of a litmus test requires comparing register values from different threads at the end of each iteration. Each thread t that performs loads has a designated array buf_t where it stores the values that were loaded into its registers in each iteration for later analysis.

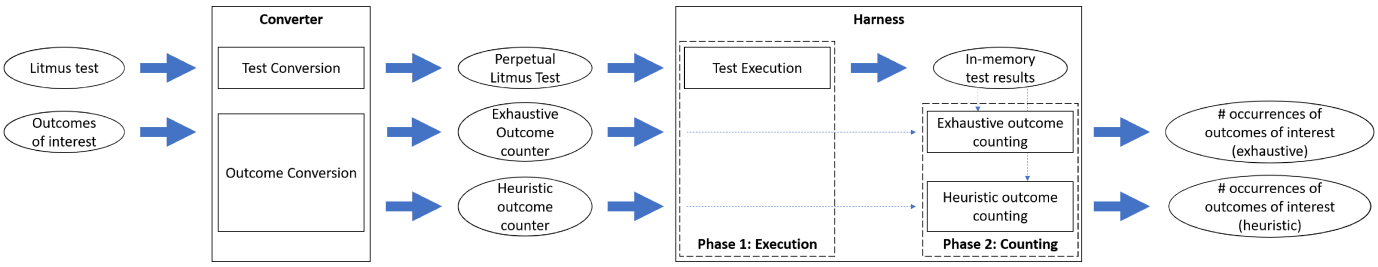

Fig. 3: PerpLE control flow diagram for generation, execution and analysis of perpetual litmus tests.

Testing tools do not launch the next iteration of the test on any thread until this information has been collected, to avoid overwriting the registers of interest.

For these reasons, a synchronization barrier is enforced across test threads, as seen on the left side of Figure 4 for the sb test from Figure 2. The barrier is a blocking call which guarantees that the litmus test threads start executing each iteration of the test simultaneously.

B. Removing barriers using arithmetic sequences

To address the overhead created by the barrier described above, we introduce perpetual litmus tests, which maintain the core approach of litmus tests, but remove per-iteration thread synchronization. Perpetual litmus tests synchronize test threads upon launch, but then do not synchronize them again until all iterations are complete. Since synchronization no longer keeps the test threads in lockstep across iterations, each test thread can run behind or ahead of the others. However, as long as we execute a large number of test iterations consecutively, each iteration of a given test thread has the chance to interact with *some* iteration of another test thread. In this setting, storing the same integer value to memory in each test iteration would be ambiguous, since multiple different store operations (from different iterations) might have stored the same value. As such, we cannot reason about the ordering of specific instructions based on loading that value. To address this challenge, we need to guarantee the *uniqueness* of each stored value across the entire run of a perpetual test. We achieve this using arithmetic sequences.

With perpetual litmus tests, each integer value that appears in a store operation of the original litmus test is mapped to a monotonically increasing arithmetic sequence of values. This lets us distinguish between stores from different iterations and eliminate ambiguity. In particular, storing a positive integer value a to a shared memory location $[mem]$ is replaced by storing an element of the arithmetic sequence $k_{mem} \cdot n_t + a$. Here, k_{mem} is the number of different integer values that appear in store operations to *mem* across all threads of the original litmus test. The iteration index n_t, which starts at 0, specifies which element of the arithmetic sequence should be written at each iteration of thread t.

For example, Figure 4 shows this conversion for the sb test, where n, m are the iteration indices of threads 0, 1 respectively. Since only the value 1 is stored to $[x]$ by an instruction of the

Litmus Test Loop Body (sb)

Initially [x] = 0, [y] = 0	
Thread 0	**Thread 1**
barrier	*barrier*
(i_{00}): [x] ← 1	(i_{10}): [y] ← 1
(i_{01}): reg_0_0 ← [y]	(i_{11}): reg_1_0 ← [x]
buf_0[n] ← reg_0_0	buf_1[m] ← reg_1_0
zero out [x], [y]	*zero out [x], [y]*

Perpetual Litmus Test Loop Body (sb)

Initially [x] = 0, [y] = 0	
Thread 0	**Thread 1**
(i_{00}): [x] ← n + 1	(i_{10}): [y] ← m + 1
(i_{01}): reg_0_0 ← [y]	(i_{11}): reg_1_0 ← [x]
buf_0[n] ← reg_0_0	buf_1[m] ← reg_1_0

Fig. 4: Store buffering (sb) litmus test and its perpetual version. n and m are iteration indices for threads 0 and 1.

Line of litmus test	Line of perpetual litmus test
$[mem] \leftarrow a, a \in \mathbb{Z}^+$	$[mem] \leftarrow k_{mem} \cdot n_t + a$
$reg_t \leftarrow [mem]$	$reg_t \leftarrow [mem]$
any fence	the same fence
(iteration end) $buf_t \leftarrow reg_t, \forall reg_t$	*(iteration end)* $buf_t \leftarrow reg_t, \forall reg_t$

TABLE I: Perpetual litmus test conversion paradigm.

original test, in particular (i_{00}), k_x is 1 and so the corresponding instruction (i_{00}) of the perpetual test now stores the value $n+1$. Similarly, storing 1 to $[y]$ in (i_{10}) has been replaced by storing $m+1$, since k_y is also 1. Thus, by leveraging arithmetic sequences, we can express each stored value as a function of the iteration that stored it. This means we also do not need to reset shared memory locations to 0 at the end of each iteration, since we can distinguish previously existing from newly stored values.

Note that load operations or fences appearing in the original litmus tests can remain unchanged. In particular, loads can proceed as usual because we still use the shared memory locations and thread registers used by the original litmus test, while fences within the test itself will have the same effect they had in the original litmus test.

Additionally, perpetual litmus tests still need access to all values loaded into registers during testing, in order to determine test outcomes at the end of the run. To achieve that, we keep the *buf* arrays of the original approach, maintaining the same storage complexity of $N \cdot T_L$. The size of each buf_t varies depending on the number of loads per iteration executed by thread t. For example, for a run of N iterations indexed by n, if thread t performs r_t load operations per iteration into registers reg_t^0 through $reg_t^{r_t-1}$ respectively, buf_t will need to be of size $r_t N$ and we will save the value of each reg_t^i into $buf_t[r_t n + i]$.

IV. Outcome Conversion and Analysis of Perpetual Litmus Tests

A. Exhaustive Outcome Counter

After removing per-iteration synchronization, we cannot determine test outcomes from the contents of the *buf* arrays in the same way as existing approaches do, since the relative timing of threads can now vary across iterations. As Figure 5 shows, for iteration n of thread 0, we can no longer simply consider its interaction with the same iteration n in other threads, since those iterations might have happened temporally far from each other. Instead, we must examine the interaction of each iteration of each thread with every iteration of every other thread, since they could happen concurrently. We define the term *frame* to refer to such a tuple of T_L iterations, one per load-performing thread, where iteration indices need not be the same. We use only load-performing threads, since their *buf* arrays hold all the information needed to determine the test outcome. Examining all frames therefore has time complexity N^{T_L} for a run of N iterations.

1) Detecting outcomes without per-iteration synchronization: To examine the way that threads interacted for a given frame, we need to express the outcomes of the original litmus test as *perpetual outcomes*, a format that takes the use of arithmetic sequences and the independent running of each thread into account. The Converter performs this using the following steps, which Figure 6 shows for each distinct outcome of the sb test:

1) Determine the happens-before edges [8] for the original litmus test outcome.
2) Replace all registers with accesses into the appropriate locations within the *buf* arrays, using a different iteration index for each thread in order to cover all frames.
3) Replace all integer values to account for the use of arithmetic sequences. Any integer value in the original outcome is loaded from shared memory, so it must have originated in some store operation (including the initializing store of 0). Thus, select a generic member of the arithmetic sequence now used by that store operation.
4) Different iterations of the same store instruction are connected with *ws* edges in iteration order. Since happens-before edges are transitive as temporal relations, we have:
   - *fr* edges: Some load L must have happened before some store S. But, L might also have happened before an even earlier store to the same location. So, L can load any term of the appropriate sequence *smaller than* that stored by S.
   - *rf* edges: Some load L must have happened after some store S. But, L might also have happened after an even later store to the same location. So, L can load any term of the appropriate sequence *larger than or equal to* that stored by S.

As Figure 6 shows, after all 4 steps have been performed on an outcome o of the original litmus test, we arrive at the corresponding perpetual outcome, which forms the body of a function p_out_o that the Converter defines. Provided with each load-performing test thread's iteration index and *buf* array, p_out_o returns true if and only if the conditions for the corresponding perpetual outcome are satisfied. The Converter repeats this process for each of the O outcomes of interest, creating the functions p_out_0 through p_out_{O-1}, each capable of detecting if the corresponding perpetual outcome occurred in a given frame.

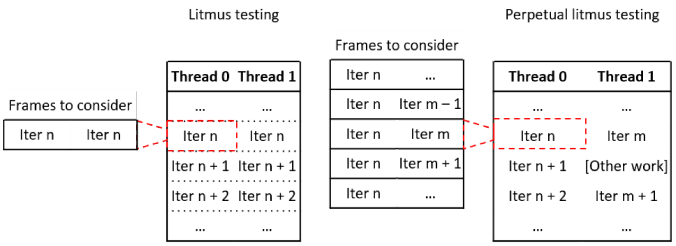

Fig. 5: To determine the outcomes in a run of perpetual tests, we examine all frames, each made up of one iteration from each thread.

Original Outcome	$reg_0_0 = 0$ && $reg_1_0 = 0$	$reg_0_0 = 0$ && $reg_1_0 = 1$
1) Happens-before edges	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{01} \rightarrow i_{10}$ (*fr*), $i_{11} \rightarrow i_{00}$ (*fr*)	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{01} \rightarrow i_{10}$ (*fr*), $i_{00} \rightarrow i_{11}$ (*rf*)
2) Replace registers	$buf_0[n] = 0$ && $buf_1[m] = 0$	$buf_0[n] = 0$ && $buf_1[m] = 1$
3) Replace integer values	$buf_0[n] = m$ && $buf_1[m] = n$	$buf_0[n] = m$ && $buf_1[m] = n + 1$
4) Turn to inequalities	***$p_out_0(n, m, buf_0[], buf_1[]) =$*** $buf_0[n] <= m$ && $buf_1[m] <= n$	***$p_out_1(n, m, buf_0[], buf_1[]) =$*** $buf_0[n] <= m$ && $buf_1[m] >= n + 1$

Original Outcome	$reg_0_0 = 1$ && $reg_1_0 = 0$	$reg_0_0 = 1$ && $reg_1_0 = 1$
1) Happens-before edges	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{10} \rightarrow i_{01}$ (*rf*), $i_{11} \rightarrow i_{00}$ (*fr*)	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{10} \rightarrow i_{01}$ (*rf*), $i_{00} \rightarrow i_{11}$ (*rf*)
2) Replace registers	$buf_0[n] = 1$ && $buf_1[m] = 0$	$buf_0[n] = 1$ && $buf_1[m] = 1$
3) Replace integer values	$buf_0[n] = m + 1$ && $buf_1[m] = n$	$buf_0[n] = m + 1$ && $buf_1[m] = n + 1$
4) Turn to inequalities	***$p_out_2(n, m, buf_0[], buf_1[]) =$*** $buf_0[n] >= m + 1$ && $buf_1[m] <= n$	***$p_out_3(n, m, buf_0[], buf_1[]) =$*** $buf_0[n] >= m + 1$ && $buf_1[m] >= n + 1$

Fig. 6: Mappings of all the outcomes of the sb test to the corresponding perpetual outcomes, based on Figure 4.

2) Counting perpetual outcome occurrences: Now that the outcomes are in a format capable of being applied to any frame, the Converter generates the exhaustive outcome counter function following the general format of Algorithm 1. The Converter replaces each reference to a *p_out* function in this generic algorithm with a specific function generated through the process above for some outcome of interest. At a high level, *COUNT* is given the number of iterations N and the *buf* arrays, which include the in-memory results of a test run. It goes through all the frames and, for each frame, evaluates each of the *p_out* functions. When one of the *p_out* functions

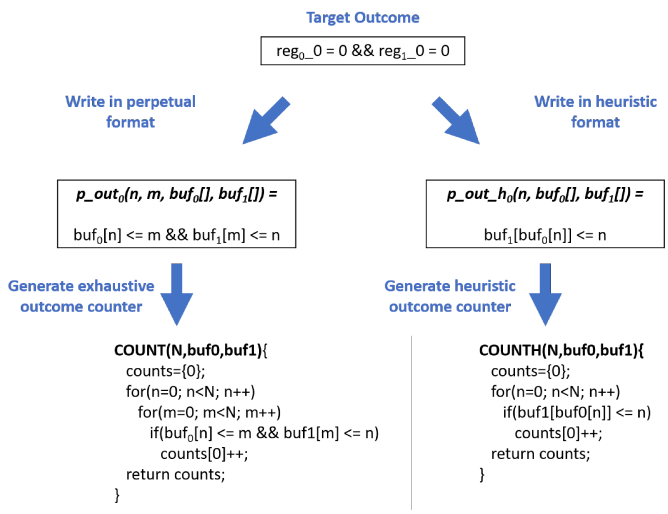

Fig. 7: Example outcome conversion and generation of the exhaustive outcome counter and heuristic outcome counter for the target outcome of the sb test.

evaluates to true, *COUNT* increments the corresponding entry in *counts*, with one entry for each perpetual outcome of interest. Note that up to one entry of *counts* is incremented for every frame.

The left part of Figure 7 shows the generation of the exhaustive outcome counter function for the target outcome of the sb test. After p_out_0 is defined for sb (top left of Figure 6), the Converter generates a version of the *COUNT* function where the reference to p_out_0 is replaced by this definition.

Algorithm 1 Exhaustive Outcome Counter

```
function COUNT(N, buf_0[], ..., buf_{T_L-1}[])

    // Initialize array of occurrences of outcomes of interest
    counts[O] = {0, ..., 0}

    // Loop through all frames, n_i is the index of thread i
    for (n_0 = 0; n_0 < N; n_0++) do
        for ... do
            for (n_{T_L-1} = 0; n_{T_L-1} < N; n_{T_L-1}++) do

                // If an outcome of interest occurred, count it
                if p_out_0(n_0, ..., n_{T_L-1}, buf_0, ..., buf_{T_L-1}) then
                    counts[0]++
                else if ... then
                    ...
                else if p_out_{O-1}(n_0, ..., n_{T_L-1}, buf_0, ..., buf_{T_L-1})
    then
                    counts[O-1]++

    return counts
```

B. Heuristic Outcome Counter

The *COUNT* function can detect all occurrences of each outcome of interest in a given test run. However, given that *COUNT*'s complexity is N^{T_L}, it can be slow when the number of iterations or test threads is large. To address this, we have developed a heuristic version of *COUNT* called *COUNTH* that has linear complexity. This function does not examine all frames, so we miss some opportunities to observe outcomes of interest. Still, we experimentally find that outcomes of interest continue to be detected more frequently than when using the traditional litmus testing approach, while the sharp decrease in runtime ultimately provides an attractive target outcome detection rate, as presented in Section VII.

The intuition behind this heuristic is as follows: because of the use of arithmetic sequences, any value loaded from shared memory indicates the iteration in which it was stored. For example, assume thread t loads the value val from the shared location $[x]$ in iteration n. This value must have been written to $[x]$ by a store in some iteration m of some thread s. We can use our knowledge of the arithmetic sequences used by stores of thread s to determine m from val. Then, m is the most recent iteration of thread s, from the point of view of iteration n of thread t. Because of this proximity in time between iteration n of thread t and iteration m of thread s, examining the frame containing n and m can be more insightful than the frame containing e.g. $n+100$ and $m-100$, two iterations which happened farther away in time and are therefore less likely to have interleaved memory operations.

In order to generate the heuristic condition for each outcome, the Converter repeats steps 1-3 of the outcome mapping procedure from Section IV-A, but it then only performs step 4 for one of the outcome conditions. A new step is then added, step 5, in which we use the remaining conditions to substitute terms of the condition we have turned into an inequality. For example, in the top left of Figure 8, which considers the sb test outcomes, we use $buf_0[n] = m$ to replace m in the inequality $buf_1[m] <= n$ with $buf_0[n]$, yielding $buf_1[buf0[n]] <= n$. By repeating this process for each outcome of interest, we obtain the functions $p_out_h_0$ through $p_out_h_{O-1}$. Each of these functions only depends on a single iteration index, since we eliminated the rest by the substitutions in step 5.

The Converter then generates the heuristic outcome counter function in a way similar to the exhaustive outcome counter function. Specifically, Algorithm 2 shows the general format of the heuristic outcome counter function. The Converter replaces each reference to a p_out_h function in this generic algorithm with a specific function generated through the modified 5-step process described above for some perpetual outcome of interest. *COUNTH* loops through the values of the remaining iteration index and, for each of them, evaluates each of the p_out_h functions. When one of the p_out_h functions evaluates to true, *COUNTH* increments the corresponding entry in *counts*, which has one entry for each perpetual outcome of interest.

V. PerpLE Tool suite

We have developed a suite of tools called the Perpetual Litmus Engine (PerpLE), capable of converting a large family of litmus tests and their outcomes to their perpetual counterparts, as well as of executing them on x86 processors. Since test conversion only needs to happen once per litmus test

Original Outcome	reg_0_0 = 0 && reg_1_0 = 0	reg_0_0 = 0 && reg_1_0 = 1
...	...	...
4) Turn to inequalities	$buf_0[n]$ = m && $buf_1[m]$ <= n	$buf_0[n]$ = m && $buf_1[m]$ >= n + 1
5) Substitute	***$p_out_h_0(n, buf_0[], buf_1[])$ =*** $buf_1[buf_0[n]]$ <= n	***$p_out_h_1(n, buf_0[], buf_1[])$ =*** $buf_1[buf_0[n]]$ >= n + 1
Original Outcome	reg_0_0 = 1 && reg_1_0 = 0	reg_0_0 = 1 && reg_1_0 = 1
...	...	...
4) Turn to inequalities	$buf_0[n]$ = m + 1 && $buf_1[m]$ <= n	$buf_0[n]$ = m + 1 && $buf_1[m]$ >= n + 1
5) Substitute	***$p_out_h_2(n, buf_0[], buf_1[])$ =*** $buf_1[buf_0[n] - 1]$ <= n	***$p_out_h_3(n, buf_0[], buf_1[])$ =*** $buf_1[buf_0[n] - 1]$ >= n + 1

Fig. 8: Heuristic condition generation for the perpetual outcome corresponding to each outcome of the sb test. By comparing to Figure 4, we see that Step 4 is not performed for the conditions in red, which are then used for substitutions.

Algorithm 2 Heuristic Outcome Counter

```
function COUNTH(N, buf_0[], ..., buf_{T_L-1}[])

    // Initialize empty array of perpetual outcome counts
    counts[O] = {0, ..., 0}

    for (n = 0; n < N; n++) do

        // If an outcome of interest occured, count it
        if p_out_h_0(n, buf_0[], ..., buf_{T_L-1}[]) then
            counts[0]++
        else if ... then
            ...
        else if p_out_h_{O-1}(n, buf_0[], ..., buf_{T_L-1}[]) then
            counts[O-1]++

    return counts
```

and outcome conversion only once for each set of outcomes of interest, we have organized PerpLE into 2 distinct tools, as shown in Figure 3. The *Converter* deals with test and outcome conversion, while the *Harness* can use the outputs of the Converter to run tests and count the occurrences of the perpetual outcomes of interest.

A. Test Conversion

The Converter, implemented in Python, receives as inputs a litmus test and a set of outcomes of interest in the format used by the litmus7 suite [11]. It then generates the corresponding perpetual test by following the strategy outlined in Section III, as well as the exhaustive and heuristic outcome counters for the outcomes of interest through the process described in Section IV. The Converter outputs the following:

- One assembly file per test thread, including that thread's instructions for the perpetual test enclosed in a loop construct, together with additional set-up and clean-up instructions as needed to handle arithmetic sequences.
- Two C files, with the exhaustive and heuristic outcome counters for this test, respectively. As per Section IV, the exhaustive outcome counter file includes $COUNT$ after replacing the functions p_out_0 through p_out_{O-1} with their definitions for the outcomes of interest. The heuristic outcome counter file includes $COUNTH$ after replacing the functions $p_out_h_0$ through $p_out_h_{O-1}$ with their definitions for the outcomes of interest.
- An additional file with the parameters t_0_reads through t_{T-1}_reads, corresponding to the number of loads that each of the T test threads performs per iteration. This is required by the Harness to allocate appropriately sized buf arrays for each test thread, as per Section III.

For this work we had the Converter generate the perpetual tests in x86 assembly language, since that was the ISA of the system we used for our evaluation. However, one could easily adapt the process to different ISAs by providing the Converter with the instructions for loads, stores and fences in the corresponding assembly language.

B. Test Execution and Perpetual Outcome Counting

The Harness is a C program that runs N iterations of a perpetual test and outputs the number of occurrences of each outcome of interest. In particular, the Harness allocates a buf array for each of the T_L load-performing test threads based on the values of t_0_reads through t_{T-1}_reads. It then launches the test threads and passes N and the appropriate buf array to each. After synchronizing once at the very beginning of the run, threads run synchronization-free for N iterations.

After the end of the run, the Harness calls the exhaustive and/or the heuristic outcome counter and provides them with the buf arrays. The $counts$ arrays returned by the exhaustive and/or the heuristic outcome counter are then returned to the user, alongside runtime information both for test execution and for the outcome counting.

C. Perpetual Litmus Suite

To evaluate PerpLE, we use the PerpLE Converter to convert a comprehensive set of TSO litmus tests from the literature [37] into perpetual litmus tests, generating the *perpetual litmus suite*. Each of these tests involves between two and four threads. For each test, the PerpLE Converter generates the corresponding perpetual test, as well as exhaustive and heuristic outcome counters for the target outcome.

Not all litmus test outcomes can be converted to their perpetual equivalents. Perpetual litmus tests lack per-iteration synchronization and shared memory locations are therefore altered in an unpredictable pattern during test execution until the run of perpetual tests terminates. PerpLE can only stop to inspect the values in these shared locations *after* the end of the entire run; inspecting them earlier gives no information, since we do not know the current iteration of each thread.

However, a class of litmus tests has target outcomes that require inspection of a value stored in shared memory at the end of every iteration. Due to the mechanics of PerpLE as described above, such outcomes cannot be converted into

Perpetual Litmus suite		
Target outcome allowed by x86-TSO		
amd3 [2,2]	iwp23b [2,2]	iwp24 [2,2]
n1 [3,2]	podwr000 [2,2]	podwr001 [3,3]
rfi009 [2,2]	rfi013 [2,2]	rfi015 [3,2]
rfi017 [2,2]	rwc-unfenced [3,2]	sb [2,2]
Target outcome forbidden by x86-TSO		
amd10 [2,2]	amd5 [2,2]	amd5+staleld [2,2]
co-iriw [4,2]	iriw [4,2]	lb [2,2]
mp [2,1]	mp+staleld [2,1]	mp+fences [2,1]
n4 [2,2]	n5 [2,2]	rwc-fenced [3,2]
safe006 [2,2]	safe007 [3,3]	safe012 [3,2]
safe018 [3,2]	safe022 [2,1]	safe024 [3,2]
safe027 [4,2]	safe028 [3,2]	safe036 [2,2]
wrc [3,2]		

TABLE II: Perpetual litmus test suite for x86-TSO. The litmus tests are split into two groups based on whether their target outcome is allowed or forbidden by the x86-TSO specification. For each test we report the values of $[T, T_L]$

.

perpetual outcomes and therefore their occurrences cannot be counted using the exhaustive or the heuristic outcome counter. We have therefore refrained from including tests with target outcomes of this nature into the perpetual litmus suite.

Table II presents the perpetual litmus test suite. The suite includes 34 litmus tests generated for the x86-TSO memory model out of the 88 tests found in the original test suite. The table splits the test suite into two groups of tests, based on whether their target outcomes are allowed or forbidden by the specification of the x86-TSO memory model.

VI. Evaluation Methodology

A. Testing Environment & Tools

To evaluate PerpLE, we use an x86 computing cluster and the suite of tests presented in Table II. Experiments are run on a CentOS 7.6 Linux cluster with 32 Intel Xeon E5-2667 CPUs with two threads per core. As explained in Section II-A2, the memory consistency model of CPUs in this cluster is a TSO variant [37], so we expect to only observe target outcomes from the "Allowed" group of tests in Table II.

We evaluate PerpLE against litmus7 on this system. For litmus7, we experiment with all available thread synchronization modes [11]: the default `user`, with polling synchronization; `userfence`, which also uses memory fences to accelerate write propagation; `pthread`, which uses a pthread-based barrier; `timebase`, which relies on the architecture's timebase counter for synchronization; and `none`, where no thread synchronization is used [24]. Timebase counters are not available in some architectures (e.g. ARM).

The `none` mode is distinct from PerpLE's approach since the concept of *frames* is not utilized; iteration n of thread t_0 is only considered with respect to iteration n of thread t_1, even though they might be executed far in time from each other. We expect this to make fine-grained thread interaction more elusive in `none` compared to PerpLE.

B. Metrics of interest

1) Target Outcome Occurrences: Because perpetual outcomes are determined per frame and the number of frames is polynomial in the number of iterations (N^{T_L}), we expect to observe each particular perpetual outcome of interest in PerpLE many more times than the corresponding outcome in litmus7, simply by virtue of exploring a much larger space, as shown in Figure 5. Moreover, since PerpLE allows different types of thread interaction compared to litmus7, outcomes which appear only rarely in litmus7 may be observed more frequently. Since observing the target outcome of a test tends to be both rarer than observing other outcomes and more helpful in understanding the underlying hardware, we compare how often the target outcome is observed in each system for a given number of test iterations.

2) Testing Runtime: Since per-iteration thread synchronization dominates runtime on litmus7 in `user` mode, its removal should significantly reduce test runtime. However, the exhaustive outcome counter must examine all N^{T_L} frames in search of perpetual outcomes after a run of N iterations and T_L load-performing test threads, as opposed to the N frames examined by litmus7. This more extensive search will likely erode the speedup achieved by eliminating synchronization. In contrast, using the heuristic outcome counter should preserve a considerable speedup over litmus7, since it only examines N frames.

3) Target Outcome Detection Rate: This composite metric shows the number of times a target outcome is observed during a test run over the time taken by the run. Since the number of occurrences is expected to increase and runtime is projected to decrease when using the heuristic outcome counter, we expect a much higher target outcome detection rate for PerpLE.

4) Heuristic Outcome Counter Accuracy: To evaluate our heuristic outcome counter, we determine its accuracy for the target outcome of each of the tests in our suite. For the target outcome of each test, we run the exhaustive and the heuristic outcome counter on the in-memory test results from the same run of perpetual tests. We then check whether, whenever the target outcome was found by the exhaustive outcome counter, it was also found by the heuristic outcome counter (not necessarily the same number of times).

5) Thread Skew: Due to the lack of per-iteration thread synchronization in PerpLE, threads can run ahead of each other by a varying number of iterations. We call the difference between the index of the iteration being executed by thread t and the index of the iteration being executed by thread s the *thread skew* between t and s. The width of the distribution of thread skew values indicates the degree to which the perpetual test run deviates from what is explored with per-iteration synchronization. This deviation can enable the perpetual test to observe system behavior that the original test misses.

To measure thread skew, we use the same insight that guided the development of heuristic conditions in Section IV-B. Namely, the value loaded by thread t through a load operation in iteration n of the perpetual test run uniquely identifies a store in some iteration m of some thread s. The difference

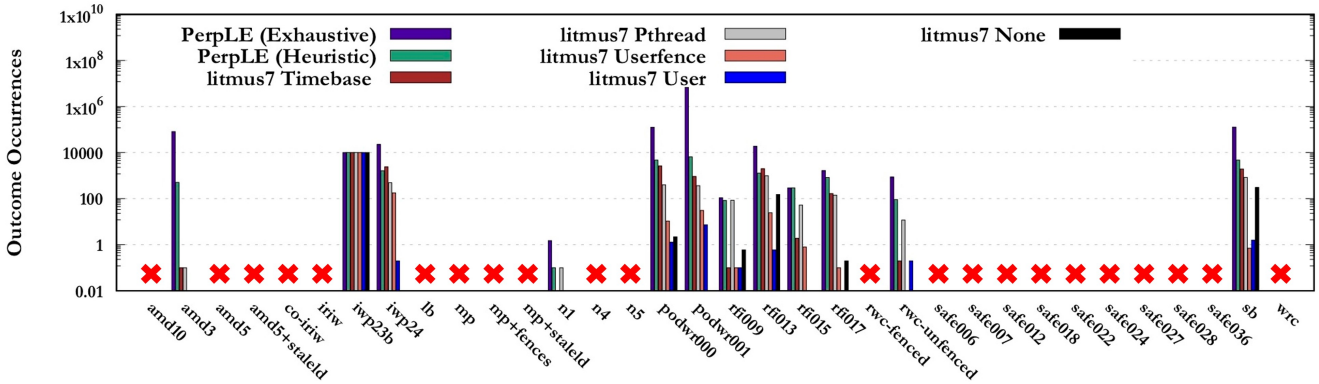

Fig. 9: Target outcome occurrences for each test of the perpetual litmus suite for 10k iterations. Higher is better. PerpLE with either outcome counter generally outperforms litmus7 in most synchronization modes. A red X symbol marks each test with a target outcome that is forbidden under x86-TSO. Note that PerpLE does not generate "false positives" in these cases. Also note that PerpLE exposes the target outcomes for all tests that are allowed under x86-TSO.

between n and m is exactly the skew between threads t and s around the time of iteration n in thread t.

6) Outcome Variety: One goal of perpetual litmus tests is to increase the effectiveness of memory consistency testing by enabling more thread interaction and generating a greater variety of test outcomes. To evaluate PerpLE with respect to this goal, we compare how frequently each possible test outcome occurs in PerpLE and in litmus7 for the same number of iterations.

VII. Evaluation

This section evaluates PerpLE using the metrics described in Section VI. In particular, we find that PerpLE (i) detects more occurrences of target outcomes, (ii) is generally superior to litmus7 when using the heuristic outcome counter, in terms of both test runtime and target outcome detection rate and (iii) provides increased outcome variety.

A. Target Outcome Occurrences

Figure 9 compares the ability of PerpLE and litmus7 in different synchronization modes to detect the target outcome of tests in the perpetual litmus suite, across 10k iterations. PerpLE with exhaustive counter performs strictly better than litmus7 in all cases, observing many occurrences of each target outcome. PerpLE with heuristic counter generally performs better than litmus7 in most cases. For the iwp24 and rfi013 tests, litmus7 in `timebase` and synchronization modes marginally outperforms PerpLE heuristic. When we increase the number of iterations beyond 10k, PerpLE is able to markedly outperform litmus7 in all synchronization modes, even for these cases.

As shown in Table II, many of the tests in the perpetual test suite have target outcomes that are forbidden under x86-TSO, as determined using the herd memory model simulator [12]; this means that we expect neither tool to observe them. As x86 CPUs have been extensively tested over the years, we can be relatively confident that our system indeed follows x86-TSO. Therefore, PerpLE's failure to observe these forbidden outcomes can be viewed as a reassurance that PerpLE does not generate false positives. In addition, PerpLE exposes target outcomes from *all* litmus tests that are allowed under x86-TSO, whereas litmus7 for certain synchronization methods fails to do so for most tests (see amd3, iwp24, n1, podwr001, rfi015, rfi017, rwc-unfenced).

B. Testing Runtime

Figure 10 presents the runtime speedup over litmus7 in `user` mode of PerpLE when using the exhaustive and heuristic outcome counter, as well as of litmus7 in the other synchronization modes. All tools execute every test in the perpetual litmus suite for 10k iterations. All runtimes include both test execution and outcome counting.

Figure 10 focuses on the comparison between PerpLE exhaustive and heuristic Counters. As discussed previously, the runtime of the PerpLE exhaustive outcome counter is polynomial to the number of test threads that perform loads, due to our examination of each frame. Most tests in our suite have two threads that perform loads, so examining a quadratic number of frames makes PerpLE with the exhaustive outcome counter significantly slower than the heuristic counter in these cases. The perpetual litmus tests that present the exhaustive outcome counter's performance comparable to the heuristic counter, e.g. mp, only have a single test thread performing loads, making the exhaustive outcome counter linear. Finally, the podwr001 and safe007 tests have three threads performing loads, so the exhaustive outcome counter needs to examine N^3 frames for a run of N iterations, yielding a dramatic slowdown. As a result the geometric average speedup of the heuristic outcome counter over the exhaustive outcome counter is 305x.

Therefore, if only focusing on runtime, using the PerpLE exhaustive outcome counter scales poorly as the number of iterations increases. The heuristic outcome counter scales much better, always taking time linear in the number of test iterations. As such, PerpLE exhaustive outcome counter performance constraints make it impractical and the remaining evaluation will therefore focus on the heuristic outcome counter. In subsequent text, the term PerpLE refers to PerpLE-heuristic.

As Figure 10 shows, when using the heuristic outcome counter, PerpLE provides a (geometric) average speedup of 8.89x over litmus7 in the default `user` mode and 17.56x, 8.85x, 2.52x and 161.35x over the `timebase`, `userfence`, `none` and `pthread` modes respectively. Note that the runtime of PerpLE with the heuristic outcome counter is comparable to `none`, since both use no synchronization and examine a linear number of frames. However, PerpLE offers significantly better outcome variety, as presented in Figure 13.

C. Target Outcome Detection Rate

To determine PerpLE's overall efficiency, we compare the target outcome detection rate between PerpLE using the heuristic outcome counter and litmus7 in different synchronization modes. For the comparison between methods we evaluated different options for averaging target outcome detection rates. Averaging outcome detection rates across different tests would implicitly skew the average towards the tests that intrinsically observe higher numbers of target outcomes. Therefore, we determine PerpLE's relative detection rate improvement by dividing PerpLE's detection rate for a given litmus test by the detection rate of litmus7 in default `user` mode for the same test; by using these ratios when averaging across all litmus tests, we avoid the aforementioned skewing problem. As we are reasoning about ratios of detection rates, we conservatively omit test cases where the baseline testing method is zero (i.e. no outcomes were detected) and provide additional details about the number of outcomes PerpLE detects in these cases.

Figure 11 presents the average relative target outcome detection rate improvement. We run experiments for 100 iterations (not shown) to 100M iterations. Each bar corresponds to the arithmetic mean of the relative improvement across all tests of the perpetual litmus suite that have target outcomes allowed in x86. PerpLE detection rate is nonzero for all allowed target outcomes for all test iteration numbers except *n1* litmus test for 100 iterations. The litmus7 `user` mode is zero (i.e. does not detect any outcomes) for all litmus tests for 100 iterations and becomes nonzero for all tests only after 1M iterations. The remaining litmus7 synchronization modes either require a similarly high number of iterations to become nonzero for all tests or never achieve that. PerpLE demonstrates the ability to discover target outcomes at low iteration counts, which litmus7 generally fails to do. More specifically, for 100 iterations (not shown), PerpLE heuristic counter and litmus7 `pthread` mode are the only tools with a non-zero detection rate.

Additionally, PerpLE is able to provide a target outcome detection rate that is strictly higher than any of litmus7's synchronization modes. For 10k iterations, PerpLE's average relative outcome detection rate improvement is between 24x (over `timebase`) and 31000x (over `user`). PerpLE is able to scale gracefully, maintaining a high relative target outcome detection rate improvement for high iteration counts: between 1800x-140000x for 10M and 1200x-44000x for 100M iterations. Overall, the target outcome detection rate of PerpLE is at least four orders of magnitude higher than that of litmus7 in the `user` (default) synchronization mode for all iteration counts.

D. Heuristic Accuracy

Figure 9 also indirectly showcases the perfect accuracy of PerpLE's heuristic outcome counter, since it tracks the exhaustive outcome counter in terms of whether the target outcome was found or not.

E. Thread Skew

Figure 12 presents a probability density function of the skew between the two threads participating in the perpetual sb litmus test, as defined in Section VI. Thread skew is a result of numerous system factors, like operating system scheduling and small differences in the time when each thread starts executing. The distribution is very wide, indicating threads can run far behind or ahead of each other. Still, it is denser around 0, since system factors might delay either test thread during execution and these effects then cancel out.

A wide range of skew values contributes to the success of perpetual litmus tests, since it is indicative of the potential for interesting cross-iteration interleavings. In contrast, traditional litmus tests are limited by synchronization and the only interleavings possible across different threads are between operations of the same iteration (no skew).

F. Outcome Variety

Figure 13 plots the number of occurrences of each outcome for the sb, lb and powdr001 litmus tests over 1k iterations. The three tests are shown in Figure 2. Litmus7 across different synchronization modes and PerpLE using the heuristic outcome counter are evaluated in terms of (i) ability to observe a large variety of outcomes and (ii) high number of observations of each individual outcome. All outcomes presented are observable under x86-TSO except for $reg_0_0 = 1$ && $reg_1_0 = 1$ in the lb litmus test (lb outcome 11 on Figure 13), which is forbidden.

Litmus7 only observes outcomes sb 11 in the `timebase` mode and powdr001 111 in the `timebase` and `userfence` modes for 1k iterations. When running 1M iterations instead, these two outcomes are observed in other litmus7 synchronization modes as well, which shows that PerpLE heuristic is capable of observing outcomes of interest using a much smaller number of test iterations. Moreover, compared to litmus7, the number of the occurrences of each outcome observed when using PerpLE heuristic is typically higher than litmus7 synchronization methods.

As per Table II, $T_L = 2$ for sb and lb, while $T_L = 3$ for podwr001. Therefore, for podwr001 we examine N^3 frames, compared to N^2 for the other two tests presented, which explains why PerpLE is able to determine an increased total number of perpetual outcomes. The heuristic for each outcome evaluates N samples out of the N^2 or N^3 available frames. It is important to note that for litmus7 the total number of occurrences for each test equals the number of test iterations, spread across the observable outcomes.

With the exception of `timebase` mode, where the two tools' results are comparable, PerpLE provides a better outcome variety than litmus7 for the same number of iterations with higher numbers of outcome occurrences.

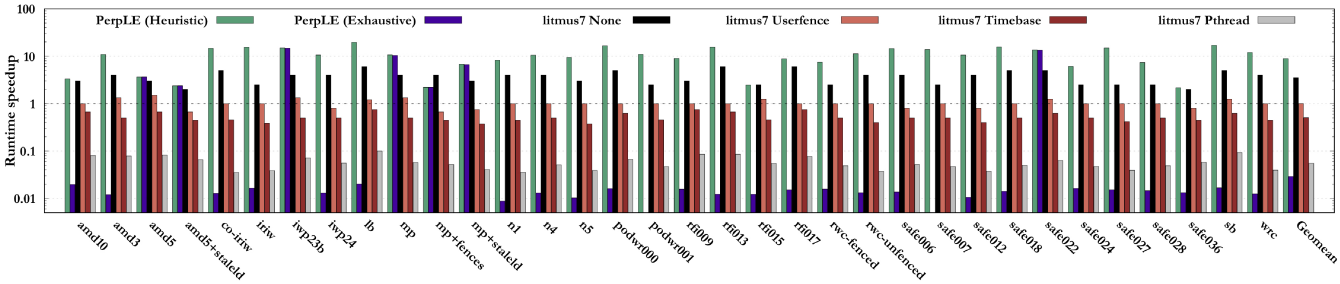

Fig. 10: Relative speedups compared to litmus7 in `user` mode (=1). We compare PerpLE Exhaustive, PerpLE Heuristic, and several litmus7 modes, for each test in the perpetual litmus suite. All runtimes include both test execution and outcome counting. Higher is better. PerpLE heuristic is always fastest.

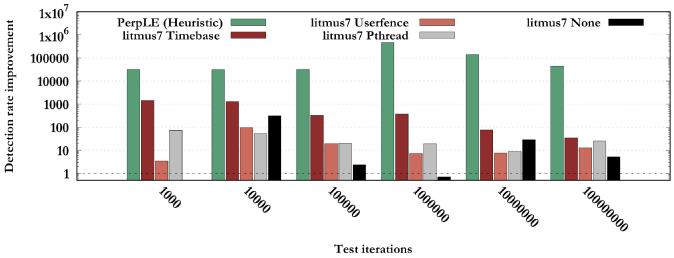

Fig. 11: Relative target outcome detection rate improvement of PerpLE using the heuristic outcome counter and litmus7 in different synchronization modes, for different numbers of test iterations. litmus7 `user` mode is the baseline.
Bars represent the arithmetic mean of the relative target outcome detection rate improvement across all perpetual litmus tests with target outcomes allowed by x86-TSO. Higher is better. PerpLE outperforms litmus7 in all modes by one to five orders of magnitude. Unlike PerpLE, litmus7 in most modes does not observe the target outcome for small iteration counts (baseline litmus7 `user` is zero for fewer than 1000 iterations).

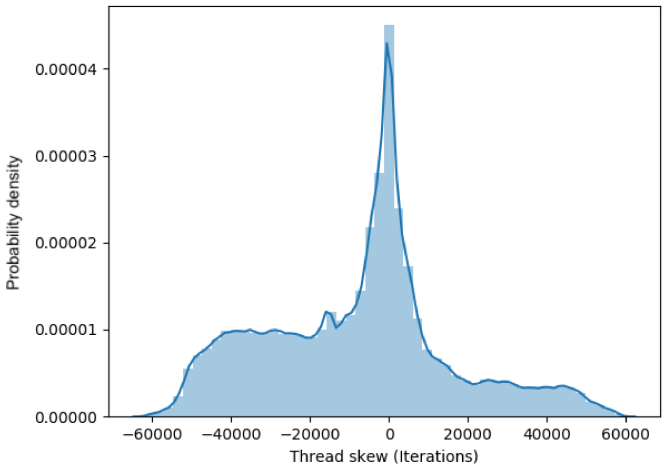

Fig. 12: Probability density of the thread execution skew (in iterations) between the two threads for 100k iterations of the store buffering (sb) perpetual litmus test. Store buffering (sb) is used as an illustrative example; other tests exhibit similar thread skew results based on our experiments.

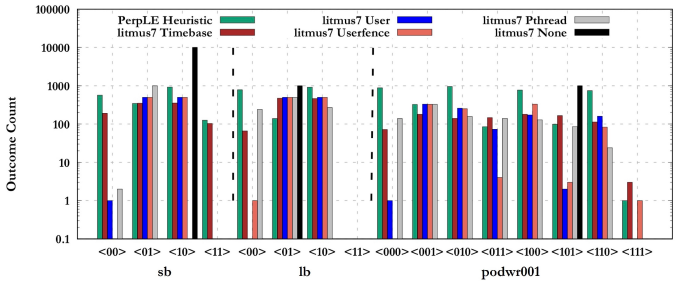

Fig. 13: PerpLE using the heuristic outcome counter and litmus7 in different synchronization modes compared in terms of outcome variety for the sb, lb and podwr001 litmus tests, for 1k iterations. PerpLE heuristic samples 1k frames *per outcome*. Higher outcome variety is better. The 11 outcome for lb is forbidden in x86-TSO.

G. Overall Impact on Testing

PerpLE provides runtime acceleration and relative detection rate improvement benefits for the overall consistency testing process. PerpLE can accelerate some important tests and notify the user about non-convertible tests, which can still be run using litmus7. When running each of the 88 tests for 10k iterations, it is 1.47x faster to use PerpLE instead of litmus7 for the 34 convertible tests (and litmus7 in `user` mode for the rest). For tests with target outcomes allowed in x86-TSO, we achieve an average relative target outcome detection rate improvement of over 20000x by using PerpLE instead of litmus7 for the convertible tests for 10k iterations.

VIII. Related Work

Memory models range widely, from sequential consistency (SC) [28] to weak memory consistency models in modern architectures [1, 2, 7, 16, 21]. Several previous works have focused on formalizing the memory consistency models of different architectures, e.g. x86 [37], POWER or more generic models [10, 12, 15], while others have focused on language level formalization of memory models [26, 27, 42]. Frequently, such efforts resort to running litmus tests on hardware and comparing their empirical observations to the published memory model specifications [6, 36]. Other efforts use random

testing instead, which can also be very effective for detecting deviations from the published model, at the cost of having a consistent test suite [18, 22, 40]. Using perpetual litmus tests can accelerate such efforts on newly developed architectures while also exposing a greater variety of outcomes, giving a fuller picture of the capabilities of the underlying hardware.

Other work has focused on specifying and verifying microarchitectural implementations of consistency models using happens-before graphs [30, 33, 34, 35, 41]. These tools focus on design time verification whereas PerpLE performs runtime evaluation of litmus tests against the hardware specification.

Based on formal memory consistency models, other efforts have addressed litmus test *generation*, for the purposes of comparing memory models or for the empirical conformance testing of new implementations of an architecture in hardware [18, 22, 32, 43]. The Converter tool in PerpLE extends such tools by converting newly generated litmus tests to their perpetual counterpart, providing automatic access to the benefits of running tests without per-iteration synchronization.

Another existing line of research has been concerned with developing tools for running non-deterministic tests, including litmus tests [5, 9, 17]. Central among these is the diy suite of tools, which includes the litmus7 tool used in our evaluation [24]. PerpLE adds to such tool development efforts, by providing a critical twist (removal of per-iteration synchronization) on a familiar approach (litmus tests). A key difference between PerpLE and litmus7 pertains to frames. Namely, PerpLE not only allows longer-term cross-iteration interleaving of events, which enriches the event orderings considered, but it also implements the logging needed to properly see these interactions from the results. Litmus7's different synchronization modes may allow for some of the same orderings, but that tool does not have the logging to see cross-iteration interleavings. Furthermore, litmus7 cannot automatically generate perpetual litmus tests from original tests, and its synchronization modes cannot enable analysis methodologies that use frames similar to PerpLE. PerpLE includes automatic test generation from original tests.

Finally, past work has been concerned with developing techniques to increase the effectiveness of litmus tests by creating different system environments for the test threads, in the hopes of exposing otherwise rare outcomes. Such approaches can have a dramatic impact: for example, Sorensen et. al [39] shows that the use of stressing and fuzzing can increase the occurrence rate of the target outcome in the lb, sb and mp litmus tests in GPUs. PerpLE also creates unusual (compared to traditional approaches) conditions for the test threads by enabling longer stretches of synchronization-free execution by each thread. The thread skew generated this way can be valuable in exercising the system, as our results show.

IX. Conclusions

Given parallelism's centrality in computing today, memory consistency testing is critical to ensure that systems and applications adhere to their formal specifications. However, current empirical litmus testing approaches waste most of the testing time waiting for threads to synchronize, a requirement that also can hurt the variety and types of outcomes of the tests. In response, we propose perpetual litmus tests, a litmus test variant that allows for consistency testing without per-iteration synchronization, by tracing happens-before edges between load and store operations using unique arithmetic sequences. We present PerpLE, a set of tools to generate, execute and analyze perpetual litmus tests and their outcomes.

PerpLE is evaluated on an x86 system, showing both greater outcome variety and more occurrences of the outcomes of interest. PerpLE can use a polynomial algorithm or an efficient, linear heuristic to identify outcomes of interest. The highly accurate heuristic provides a significant speedup, leading to an overall target outcome detection rate that is orders of magnitude higher than prior state of the art. This paper focused on x86-TSO, but our approach can also be applied to architectures implementing weaker memory models. These improvements can help expand the applicability and effectiveness of empirical memory consistency testing.

Acknowledgements

We thank our shepherd, the anonymous reviewers and Tyler Sorensen for their helpful feedback. This work was supported by the U.S. National Science Foundation through the grants CNS 1739674, CNS 1739701 and CNS 2004118.

References

[1] "Alpha architecture reference manual," 1992.
[2] "SPARC architecture manual, version 9," 1994.
[3] "9th Generation Intel Core Desktop Processors," 2019. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/9th-gen-core-desktop-brief.pdf
[4] "The GeForce 16 Series Graphics Cards are Here," 2019. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/gtx-1660-ti/
[5] A. Adir and G. Shurek, "Generating Concurrent Test-programs with Collisions for Multi-processor Verification," in *Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop*, ser. HLDVT '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 77–. [Online]. Available: http://dl.acm.org/citation.cfm?id=1114283.1114492
[6] A. Adir, H. Attiya, and G. Shurek, "Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 5, pp. 502–515, May 2003. [Online]. Available: https://doi.org/10.1109/TPDS.2003.1199067
[7] S. V. Adve and M. D. Hill, "Weak ordering -a new definition," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: ACM, 1990, pp. 2–14. [Online]. Available: http://doi.acm.org/10.1145/325164.325100
[8] J. Alglave, "A formal hierarchy of weak memory models," *Formal Methods in System Design*, vol. 41, no. 2, pp. 178–210, 6 2012.
[9] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU Concurrency: Weak Behaviours and Programming Assumptions," *SIGPLAN Not.*, vol. 50, no. 4, pp. 577–591, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775054.2694391
[10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in Weak Memory Models," in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, ser. CAV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 258–272. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_25
[11] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running Tests against Hardware," *Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science*, pp. 41–44, 2011.

[12] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2627752

[13] E. Blem, J. Menon, and K. Sankaralingam, "A Detailed Analysis of Contemporary ARM and x86 Architectures," *UW-Madison Technical Report*, 2013.

[14] J. Bornholt, "Memory Consistency Models: A Tutorial," Feb 2016. [Online]. Available: https://homes.cs.washington.edu/~bornholt/post/memory-models.html

[15] W. W. Collier, *Reasoning about parallel architectures*. Prentice-Hall, Inc., 1992.

[16] F. Corella, J. M. Stone, and C. M. Barton, "A formal specification of the PowerPC shared memory architecture," *CS Tech. Report RC 18638 (81566)*, 1993.

[17] M. Diagnostics, "ARCHTEST program," 2009. [Online]. Available: http://www.mpdiag.com/archtest.html

[18] M. Elver and V. Nagarajan, "McVerSi: A test generation framework for fast memory consistency verification in simulation," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 618–630.

[19] H. Foster, "Part 8: The 2018 Wilson Research Group Functional Verification Study," January 2019. [Online]. Available: https://blogs.mentor.com/verificationhorizons/blog/2019/01/29/part-8-the-2018-wilson-research-group-functional-verification-study/

[20] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2744769.2744921

[21] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90, 1990, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/325164.325102

[22] S. Hangal, D. Vahia, C. Manovit, J. . J. Lu, and S. Narayanan, "TSOtool: a program for verifying memory systems using the memory consistency model," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, June 2004, pp. 114–123.

[23] M. D. Hill and V. J. Reddi, "Accelerator-level Parallelism," 2019.

[24] Inria, "The diy software suite," July 2019. [Online]. Available: http://diy.inria.fr/

[25] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 200–212.

[26] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A promising semantics for relaxed-memory concurrency," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017, 2017, pp. 175–189. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009850

[27] O. Lahav, N. Giannarakis, and V. Vafeiadis, "Taming release-acquire consistency," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, 2016, pp. 649–662. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837643

[28] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[31] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015, pp. 388–400. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750378

[29] C. Lin, V. Nagarajan, and R. Gupta, "Efficient Sequential Consistency Using Conditional Fences," *International Journal of Parallel Programming*, vol. 40, no. 1, pp. 84–117, 2012. [Online]. Available: https://doi.org/10.1007/s10766-011-0176-3

[30] D. Lustig, M. Pellauer, and M. Martonosi, "Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014, pp. 635–646. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.38

[32] D. Lustig, A. Wright, A. Papakonstantinou, and 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*, ser. ASPLOS '17, 2017, pp. 661–675. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037723

[33] Y. A. Manerkar, D. Lustig, M. Martonosi, and A. Gupta, "PipeProof: Automated memory consistency proofs for microarchitectural specifications," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 788–801. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00069

[34] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLcheck: Verifying the memory consistency of rtl designs," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, 2017, pp. 463–476. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124536

[35] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: Using μhb graphs to verify the coherence-consistency interface," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48, 2015, pp. 26–37. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830782

[36] L. Maranget, S. Sarkar, and P. Sewell, "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models," 2012. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf

[37] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: x86-TSO," *Lecture Notes in Computer Science Theorem Proving in Higher Order Logics*, pp. 391–407, 2009.

[38] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[39] T. Sorensen and A. F. Donaldson, "Exposing Errors Related to Weak Memory in GPU Applications," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 100–113. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908114

[40] T. Ta, X. Zhang, A. Gutierrez, and B. M. Beckmann, "Autonomous Data-Race-Free GPU Testing," in *IEEE International Symposium on Workload Characterization*, 2019.

[41] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "Tricheck: Memory model verification at the trisection of software, hardware, and isa," *SIGPLAN Not.*, vol. 52, no. 4, pp. 119–133, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093336.3037719

[42] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli, "Common compiler optimisations are invalid in the c11 memory model and what we can do about it," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, 2015, pp. 209–220. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676995

[43] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically Comparing Memory Consistency Models," in *ACM SIGPLAN Notices*, vol. 52, no. 1, 2017, pp. 190–204.