Challenges and Opportunities in Heterogeneous Parallelism

Markos Markakis

May 4, 2020

Advised by Professor Margaret Martonosi

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering.

Department of Electrical Engineering
Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

Markos Markakis

Challenges and Opportunities in Heterogeneous Parallelism

Markos Markakis

ABSTRACT

Even as most of today's computer systems have turned to parallelism to improve performance, important challenges still need to be addressed.

In Part A, we observe that documentation often remains informal, incomplete or even incorrect regarding the memory consistency models of parallel systems, leading to programmer and designer confusion and bugs. Existing tools for empirical memory consistency testing use large numbers of iterations of simple multi-threaded litmus tests, typically employing thread synchronization at every iteration. This synchronization imposes a significant overhead, reducing testing performance and efficiency. We propose new litmus test variants called *perpetual litmus tests*, which allow for more efficient consistency testing, using arithmetic sequences to reduce the required synchronization points. We present PerpLE, a software suite 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. We evaluate PerpLE against litmus7 on an x86 system. Our tool exposes a wider variety of outcomes while being faster than all litmus7 synchronization modes (8.89x faster than the default `user` mode), while the detection rate of outcomes of interest is increased by over two orders of magnitude.

In Part B, we verify the Decoupled Consumer-Producer (DCP) subsystem of the Intelligent Storage (IS) tiles in the DECADES architecture. In order to exhaustively verify DCP despite its complexity, we turn to formal verification, which we perform bottom-up in three phases. Based on the system specification, we write properties in the SVA language and use JasperGold, a formal verification tool, to mathematically prove that our design never violates them. Having completely verified a single-FIFO version of our design, we are now in Phase II of verifying the current multi-FIFO version, where we have unbounded proofs for 78% of properties and bounded proofs of between 18 and 32 cycles for the rest. JasperGold also finds 100% of our code to be reachable and all finalized code (90% of the total code) to be explicitly tested by at least one property.

Acknowledgements

Part A of this report is based on an academic paper of the same title, authored by Themistoklis Melissaris, Markos Markakis, Kelly Shaw and Margaret Martonosi, currently submitted to the 53rd IEEE/ACM International Symposium on Microarchitecture.

I would like to thank all the members of the MRM group for their incredible support and mentoring throughout both projects described in this report. First and foremost, I would like to thank my adviser, Professor Margaret Martonosi, for welcoming me to her research group and providing invaluable guidance throughout the past year and a half. Regarding Part A, I am grateful for my collaboration with Themistoklis Melissaris, without whose creativity and hard work PerpLE could not have been developed, as well as with Dr. Tyler Sorensen, with his infectious enthusiasm for parallel programming and litmus tests. Regarding Part B, my gratitude goes out to Marcelo Orenes Vera for introducing me to formal verification and providing an invaluable industry perspective on hardware design, as well as to Dr. Esin Tureci for her mentoring and thoughtful comments as this manuscript was being finalized. I would also like to thank all of the undergraduate students working with the MRM group for their contributions, questions and comments that pushed these projects forward.

Beyond our research group, I am extremely thankful for the generous support of these projects by Princeton University's Electrical Engineering and Computer Science Departments, yet another reaffirmation of their commitment to undergraduate research.

Last but certainly not least, none of my academic endeavors would have been possible without the continuous supportive presence of my family and friends, both at Princeton and back home. Thank you for being there for me through it all.

Contents

Part A

PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing[1]

1 Introduction

As traditional approaches to increasing system performance have been constrained by the end of Moore/Dennard scaling, exploiting parallelism has become the primary way to gain further performance improvements across different system types [2][3][4]. 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 [5].

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. While some comprehensive formal techniques exist [6][7], industry testing of real hardware relies more on empirical and probabilistic testing [8]. 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. In order to maximize the probability of detecting bugs, empirical testing tools aim to expose, through their outcomes, as large a variety of orderings as possible.

Currently, tools achieve this by executing litmus tests iteratively, with different orderings arising probabilistically during test execution due to factors like system load and thread timing [8]. However, each litmus test might need to be run thousands or even millions of times before the

[1] Text and figures adapted from [1].

specific orderings that the tester cares about, which are evident through what we call the *outcomes of interest*, are observed [5]. 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 favored outcomes, some testing approaches may require executing large numbers of iterations, thus also devoting 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 [8][9], are effective empirical testing frameworks, they usually 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 to allow for interesting interaction via shared memory, but it can have negative implications. To begin with, this tight synchronization might reduce the number and type of orderings that ever arise during the iterative test, contrary to the aim of exposing a large variety of orderings. At the same time, the synchronization overhead dominates runtime, significantly slowing down testing. 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 is consistently above 85% of total execution time. This can make iterative synchronization-based testing prohibitively slow 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 [10]. In fact, consistency testing in these systems is orders of magnitude slower compared to microprocessors [5].

PerpLE is designed to increase the opportunities to observe different outcomes of interest per unit time. Our approach, centered around *perpetual litmus tests*, rethinks the design of litmus tests and reduces the overall testing time by eliminating per-iteration synchronization. At the same time, letting threads run longer without synchronization creates more opportunities for interesting interactions, leading to a greater variety of orderings and, by extension, of observed outcomes. While other litmus testing tools may also provide synchronization-free modes (e.g. the `none` mode of litmus7 [8]), PerpLE offers increased testing efficiency over such 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*, which can detect a polynomially higher number of outcomes compared to existing methodologies, both synchronization-based and synchronization-free.
- We also present a linear heuristic that allows PerpLE to scale to millions of test iterations while maintaining a rate of target outcome discovery that is orders of magnitude higher than existing approaches, both synchronization-based and synchronization-free.
- We evaluate PerpLE on an x86 system, showing higher outcome variety and target outcome occurrence rate compared to all litmus7 synchronization modes, including the `none` mode.
- We find that PerpLE achieves a runtime speedup over all litmus7 synchronization modes (8.89x over `user` mode and 2.52x over the synchronization-free `none` mode).
- We also observe an increase of over two orders of magnitude in the number of outcomes of interest observed per unit time, making testing practical for high-latency parallel systems.

This work was completed in close collaboration with Themistoklis Melissaris, who is the original source of many of the central ideas of PerpLE. Even though I actively contributed to all parts of the project as we moved from the whiteboard to a complete implementation, my personal contribution was the greatest in the following areas:

- Developing the mapping from litmus test outcomes to the corresponding perpetual outcomes, presented in Section 4.1. This contribution is central in order to make the outcomes of perpetual tests interpretable and comparable to the outcomes of existing approaches.
- Developing the heuristic outcome counting algorithm, presented in Section 4.2. This contribution helps contain the combinatorial explosion caused by examining all possible frames, which could erode the performance improvements from eliminating synchronization.

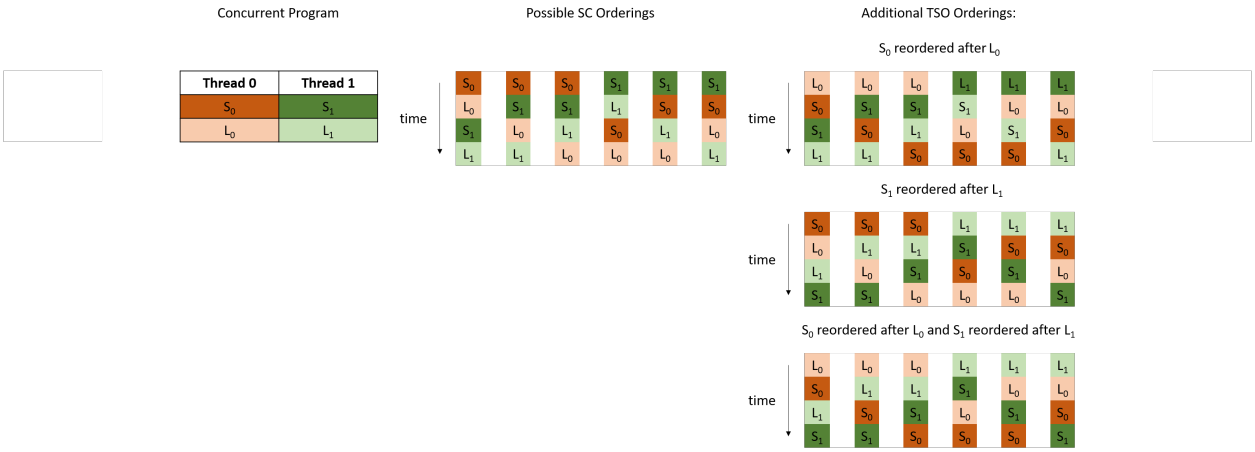

Figure 1: Possible memory operation orderings under SC and TSO. S_0 and S_1 are stores, while L_0 and L_1 are loads. Under SC, the possible orderings are interleavings of the memory operations of each thread in program order. TSO relaxes the program order requirement by allowing store buffering, so stores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

2 Background

2.1 Memory Consistency Models

2.1.1 Sequential Consistency

In the context of a multi-core system, memory consistency models are the rules that determine the ordering of concurrent loads and stores, and therefore determine which values can be legally returned to loads. In Lamport's *sequential consistency (SC)* [11], all threads see loads and stores to memory in the same global 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 middle column of Figure 1 for the concurrent program provided.

Even though sequential consistency is very intuitive, it is rarely found in real systems, because its strong guarantees limit the potential for exploiting memory system concurrency [12][13]. Modern processors usually relax the guarantee that the per-thread program order is preserved globally and turn to other weaker memory models instead.

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_2_0 ← [x]

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

2.1.2 TSO

Total Store Ordering (TSO) can be informally understood as the memory model that is obtained from SC by introducing store buffers for each thread to reduce the latency of store operations. Each thread can load the values from its own store operations directly out of its store buffer, before other threads can see them. This can provide additional allowed orderings in some cases, as shown in the rightmost column of Figure 1.

Work by Owens et al. has shown that a version of TSO is consistent with the behavior of x86 processors across a number of test cases [14][15]. As such, TSO is one of the predominant memory consistency models one encounters when analyzing the memory behavior of CPUs. Our evaluation in this work focuses on TSO, but our approach is applicable to weaker models as well.

2.2 Uncovering Instruction Orderings

2.2.1 Litmus Tests

Small stressmark tests known as *litmus tests* are used to test that software and hardware systems adhere to their specified memory models. 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, a three-thread version of sb.

Running a litmus test yields 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*. For example, the sb test from Figure 2 has 4 possible outcomes, each consisting of 2 conditions: ($reg_0\_0 = 0$ and $reg_1\_0 = 0$); ($reg_0\_0 = 0$ and $reg_1\_0 = 1$); ($reg_0\_0 = 1$ and $reg_1\_0 = 0$); and ($reg_0\_0 = 1$ and $reg_1\_0 = 1$). Note that the first of these outcomes cannot occur by simply interleaving the instructions from each thread. As such, it is the most informative outcome in terms

of hardware capabilities, letting us distinguish between SC and consistency models allowing for store buffering. Each litmus test has at least one such outcome, the *target outcome*.

After executing multiple iterations of a litmus test, the observed orderings can 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 an outcome not observed in a given run would never be observed in future runs.

In practice, available tools for litmus testing use two approaches to address this non-determinism [9][16]. First, they use runs consisting of a large number of litmus test iterations, to give a statistically more accurate picture of the distribution of outcomes. Second, tools might apply further stress to 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 [5].

2.2.2 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*. In a happens-before graph, the different shared memory operations in a litmus test are conceptualized as vertices in a graph. Edges are added to represent temporal relationships between pairs of operations, based on the outcome of a specific execution of the test. Note that since edges represent temporal relationships, they are transitive. Alglave [17] provides formal descriptions of four types of such edges between two memory operations m_1 and m_2, summarized below:

- Program order (*po*) edges: a *po* edge from m_1 to m_2 means that a sequential processor would execute 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.

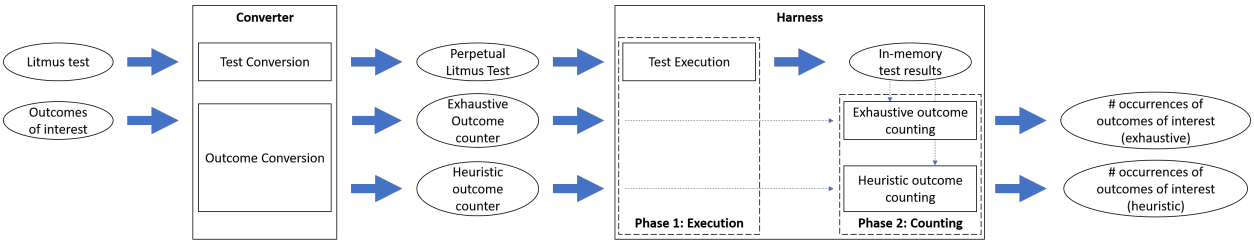

Figure 3: PerpLE control flow diagram for generation, execution and analysis of perpetual litmus tests.

3 Test Conversion to Remove Synchronization

As outlined in Figure 3, our proposed approach consists of two steps, each performed by a separate tool. First, the *Converter* converts an input litmus test in 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*. A test run involving 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 a test run has been completed, to determine how many times each outcome of interest occurred. The Converter also produces the *heuristic outcome counter*, a function with the same inputs and outputs as the exhaustive outcome counter, but which only searches part of the results space and is therefore dramatically faster.

The rest of Section 3 presents the methodology for test conversion, while Section 4 deals with generating the exhaustive and heuristic outcome counter functions.

3.1 Synchronization in Litmus Tests

Since litmus tests tend to only be a few instructions long, a thread can execute a single iteration very quickly. It appears that, unless we synchronize the participating threads at the start of each iteration, threads will most likely each execute their part of the test at different times. Thus, interactions among individual memory operations executed by different threads will be unlikely.

Moreover, there is a reset procedure that must take place after each iteration. First, determining the outcome of a litmus test requires comparing register values from different threads at the end of each iteration. In existing tools, a designated array buf_t is allocated for each thread

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

Figure 4: Store buffering (sb) litmus test and its perpetual version. n and m are the test iteration indices for threads 0 and 1 respectively. Note that the barrier would enforce $n = m$ for the case on the left.

t, to keep the values that were loaded into its registers in each iteration for later analysis. Second, the shared memory locations used by the test must also be reset to their default value (usually 0) after each iteration, to make the results of the following iterations interpretable. Testing tools do not launch the next iteration of the test on any thread until this reset procedure has been completed.

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.

3.2 Removing Barriers Using Arithmetic Sequences

The barrier described above creates an overhead, which we address by introducing perpetual litmus tests. While maintaining the core approach of litmus tests, we remove per-iteration thread synchronization, as well as the resetting of shared memory locations to their default values after each iteration, as seen in Figure 4. Both of these processes now only take place in the very beginning of a test run. Since per-iteration thread synchronization no longer keeps the test threads in lockstep, each thread can run behind or ahead of the others. However, as long as our runs are long enough, the vast majority of iterations of a given test thread will have the chance to interact with *some* iteration of another test thread. Storing fixed integer values to memory as part of the test would now be ambiguous. Since store operations from different iterations might have resulted in the same value being stored, we would no longer be able to distinguish these operations based on loading that value. To address this challenge, we need to guarantee the *uniqueness* of each stored value across all iterations of a testing run. We achieve this using arithmetic sequences.

	Store	Load	Fence	Iteration end
Litmus test	$[mem] \leftarrow a, a \in \mathbb{Z}^+$	$reg_t \leftarrow [mem]$	any fence	$buf_t \leftarrow reg_t, \forall reg_t$
Perpetual litmus test	$[mem] \leftarrow k_{mem} \cdot n_t + a$	$reg_t \leftarrow [mem]$	the same fence	$buf_t \leftarrow reg_t, \forall reg_t$

Table 1: Perpetual litmus test conversion paradigm.

With perpetual litmus tests, each fixed 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, as shown in Table 1, 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 test iteration index n_t, which starts at 0, specifies which element of the arithmetic sequence should be written at each iteration of thread t, and a is now simply an offset.

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 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 appearing in the original litmus test can remain unchanged, since the shared memory locations used by the test are still the same. The same would be true for any memory fences involved in the test itself, since they will have the same effect in the perpetual test that they had in the original litmus test.

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; we therefore keep the *buf* arrays of the original approach. 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 $buf_t[r_t n + i]$.

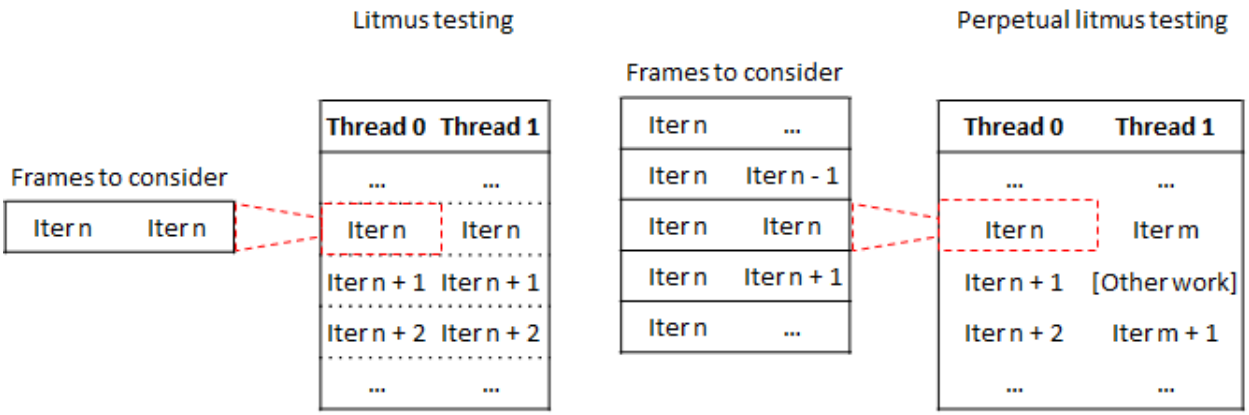

Figure 5: To determine the outcomes in a run of perpetual tests, we examine all frames, each made up of one iteration from each thread. Dotted gray lines indicate thread synchronization.

4 Outcome Conversion and Analysis of Perpetual Litmus Tests

4.1 Exhaustive Outcome Counter

When using perpetual litmus tests, test outcomes cannot be determined from the contents of the *buf* arrays in the same way as when using simple litmus tests. As Figure 5 shows, since the relative timing of threads can now vary, we can no longer simply consider the interaction of iteration n in thread 0 with the same iteration n in other threads, because might have happened temporally far from each other. Instead, we need to examine the interaction of iteration n in thread 0 with *each* iteration in other threads, since, without synchronization, any one of them could have happened concurrently. We call each set of exactly one iteration from each thread a *frame*.

4.1.1 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 into account the use of arithmetic sequences and the independent running of each thread. The Converter performs this using the following steps, which Figure 6 shows for each distinct outcome of the sb test:

1. Find the edges in the happens-before graph 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.

Original Outcome	$reg_0\_0 = 0$ && $reg_1\_0 = 0$	$reg_0\_0 = 0$ && $reg_1\_0 = 1$	$reg_0\_0 = 1$ && $reg_1\_0 = 0$	$reg_0\_0 = 1$ && $reg_1\_0 = 1$
1) Happens-before edges	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{01} \rightarrow i_{10}$ (*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*)	$i_{00} \rightarrow i_{01}$ (*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	$buf_0[n] = 0$ && $buf_1[m] = 0$	$buf_0[n] = 0$ && $buf_1[m] = 1$	$buf_0[n] = 1$ && $buf_1[m] = 0$	$buf_0[n] = 1$ && $buf_1[m] = 1$
3) Replace integer values	$buf_0[n] = m$ && $buf_1[m] = n$	$buf_0[n] = m$ && $buf_1[m] = n + 1$	$buf_0[n] = m + 1$ && $buf_1[m] = n$	$buf_0[n] = m + 1$ && $buf_1[m] = n + 1$
4) Turn to inequalities	***p_out*$_0$*(n, m, buf*$_0$*[], buf*$_1$*[]) =* $buf_0[n] <= m$ && $buf_1[m] <= n$	***p_out*$_1$*(n, m, buf*$_0$*[], buf*$_1$*[]) =* $buf_0[n] <= m$ && $buf_1[m] >= n + 1$	***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*$_1$*[]) =* $buf_0[n] >= m + 1$ && $buf_1[m] >= n + 1$

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

3. Replace 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 been stored there by a store operation. Replace it with a generic member of the arithmetic sequence used by that store operation in the perpetual test.

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:

   - For each *fr* edge, we know that 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 may as well load *any* term of the appropriate sequence *smaller than* that stored by S.
   - For each *rf* edge, we know that 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 may as well load terms of the appropriate sequence *larger than* 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 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.

4.1.2 Counting Perpetual Outcome Occurrences

Once outcomes are expressed in a format consistent with the concept of frames, 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 in Section 4.1.1. 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, one for each of T test threads. It goes through all the frames and, for each frame, evaluates each of the *p_out* functions. When one of the *p_out* functions evaluates to true, $COUNT$ increments by 1 the corresponding entry in *counts*, an array 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 in Figure 6, the Converter replaces the reference to p_out_0 in $COUNT$ by this definition and returns the resulting function.

Algorithm 1 Exhaustive Outcome Counter

```
 1: function COUNT(N, buf_0[], ..., buf_{T-1}[])
 2:
 3:     // Initialize array of occurrences of outcomes of interest.
 4:     counts[O] = {0, ..., 0}
 5:
 6:     // Loop through all frames, n_i is the index of thread i.
 7:     for (n_0 = 0; n_0 < N; n_0++) do
 8:         for ... do
 9:             for (n_{T-1} = 0; n_{T-1} < N; n_{T-1}++) do
10:
11:                 // If an outcome of interest occurred, count it.
12:                 if p_out_0(n_0, ..., n_{T-1}, buf_0, ..., buf_{T-1}) then
13:                     counts[0]++
14:                 else if ... then
15:                     ...
16:                 else if p_out_{O-1}(n_0, ..., n_{T-1}, buf_0, ..., buf_{T-1}) then
17:                     counts[O-1]++
18:
19:     return counts
```

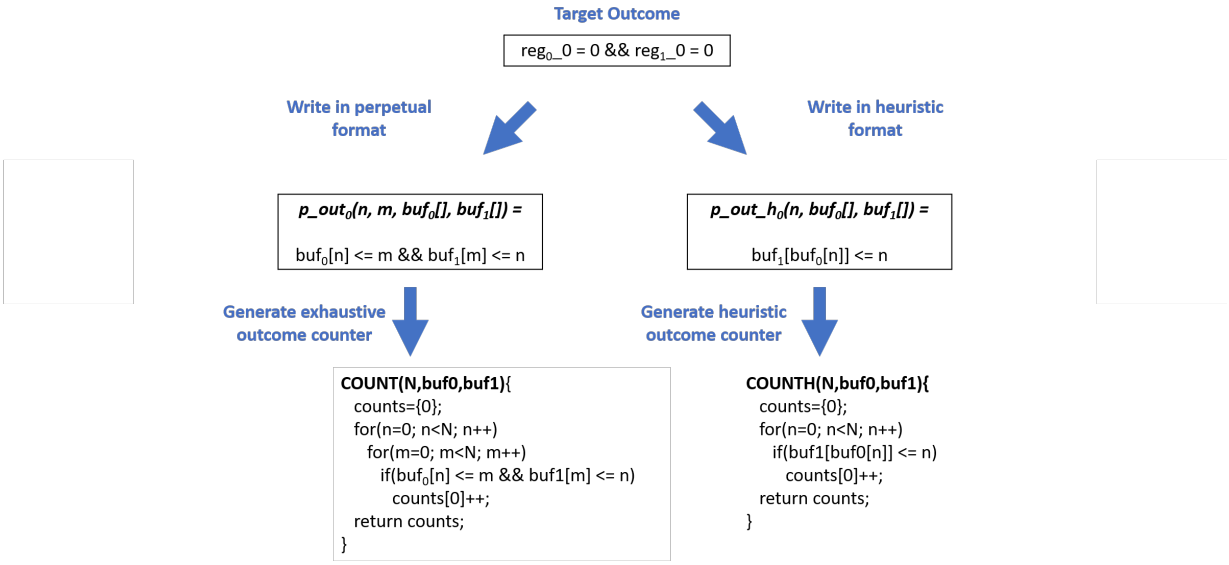

Figure 7: Example outcome conversion and generation of the exhaustive outcome counter and heuristic outcome counter for the target outcome of the sb test.

4.2 Heuristic Outcome Counter

The $COUNT$ function can detect all occurrences of each outcome of interest in a given test run, but has complexity N^T, making it slow when the number of iterations (N) or test threads (T) is large. To address this, we have developed a heuristic version of $COUNT$ called $COUNTH$ that has complexity N. By not examining all frames, $COUNTH$ misses some opportunities to observe outcomes of interest. Still, we experimentally find that $COUNTH$ detects more outcomes of interest than litmus7, while the sharp decrease in runtime ultimately provides an attractive outcome detection rate for the target outcome, as presented in Section 7.

This heuristic builds on the following observation: because of the use of arithmetic sequences, values loaded from shared memory each indicate the iteration in which they were 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. Then, m must have been the most recent iteration of thread s, from the point of view of iteration n of thread t. Knowing the arithmetic sequences used by stores of thread s, we can determine m from val. This temporal proximity between iteration n of thread t and iteration m of thread s can make the frame containing n and m more insightful than the frame containing e.g. $n+100$ and $m-100$, which happened farther away in time and are less likely to have interleaved memory operations.

Original Outcome	$reg_0\_0 = 0$ && $reg_1\_0 = 0$	$reg_0\_0 = 0$ && $reg_1\_0 = 1$	$reg_0\_0 = 1$ && $reg_1\_0 = 0$	$reg_0\_0 = 1$ && $reg_1\_0 = 1$
...	...	...	...	...
4) Turn to inequalities	$buf_0[n] = m$ && $buf_1[m] <= n$	$buf_0[n] = m$ && $buf_1[m] >= n + 1$	$buf_0[n] = m + 1$ && $buf_1[m] <= n$	$buf_0[n] = m + 1$ && $buf_1[m] >= n + 1$
5) Substitute	***p_out_h0(n, buf0[], buf1[]) =*** $buf_1[buf_0[n]] <= n$	***p_out_h1(n, buf0[], buf1[]) =*** $buf_1[buf_0[n]] >= n + 1$	***p_out_h2(n, buf0[], buf1[]) =*** $buf_1[buf_0[n] - 1] <= n$	***p_out_h3(n, buf0[], buf1[]) =*** $buf_1[buf_0[n] - 1] >= n + 1$

Figure 8: Heuristic condition generation for the perpetual outcome corresponding to each outcome of the sb test. By comparing to Figure 6, we see that Step 4 is not performed for the conditions in red, which are then used for substitutions.

In order to generate the heuristic condition for each outcome, the Converter repeats steps 1-3 of the procedure from Section 4.1.1, but then only performs step 4 for one of the conditions in each outcome. A new step is then added, step 5, in which the remaining conditions (shown in red) are used to substitute terms of the condition turned into an inequality. For example, in the first column of Figure 8, $buf_0[n] = m$ is used to replace m in the inequality $buf_1[m] \leq n$ with $buf_0[n]$, yielding $buf_1[buf_0[n]] \leq n$. By repeating this process for each outcome of interest, we get the functions $p\_out\_h_0$ through $p\_out\_h_{O-1}$, each of which only includes a single iteration index (n).

The Converter then generates the heuristic outcome counter function following the general format of Algorithm 2. 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. At a high level, $COUNTH$ 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 by 1 the corresponding entry in $counts$, an array with one entry for each perpetual outcome of interest.

5 PerpLE Tool Suite

We have developed the Perpetual Litmus Engine (PerpLE), a tool suite consisting of two tools 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. The *Converter* deals with test and outcome conversion, which only needs to happen once for each litmus test and each set of outcomes of interest, while the *Harness* can use the outputs of the Converter to run tests and count the occurrences of the perpetual outcomes of interest.

Algorithm 2 Heuristic Outcome Counter

```
1:  function COUNTH(N, buf_0[], ..., buf_{T-1}[])
2:
3:      // Initialize empty array of perpetual outcome counts.
4:      counts[O] = {0, ..., 0}
5:
6:      for (n = 0; n < N; n++) do
7:
8:          // If an outcome of interest occurred, count it.
9:          if p_out_h_0(n, buf_0[], ..., buf_{T-1}[]) then
10:             counts[0]++
11:         else if ... then
12:             ...
13:         else if p_out_h_{O-1}(n, buf_0[], ..., buf_{T-1}[]) then
14:             counts[O - 1]++
15:
16:     return counts
```

5.1 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 [9]. It then generates the corresponding perpetual test by following the approach of Section 3, as well as the exhaustive and heuristic outcome counters for the outcomes of interest through the processes described in Section 4. 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 some 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 4, 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 3.

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 the operations of interest (i.e. loads, stores, fences) in the corresponding assembly language.

5.2 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 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 subsequently 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.

5.3 Perpetual Litmus Test Suite

To evaluate PerpLE, we use the PerpLE Converter to generate a comprehensive set of TSO perpetual litmus tests based on the literature [14], generating the *perpetual litmus test suite*, as presented in Table 2. Each of these tests involves between two and four test threads. For each test, the PerpLE Converter has generated the corresponding perpetual test, as well as an appropriate exhaustive and heuristic outcome counters for the target outcome of the test.

Note that some litmus test outcomes may require inspecting a value stored in shared memory at the end of each iteration and before the start of the next one. Since perpetual litmus tests lack per-iteration synchronization, threads can be arbitrarily out-of-sync, which makes timing this inspection impossible. Due to this fact, such outcomes cannot be converted into 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 test suite.

Perpetual Litmus Test Suite

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

Table 2: 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.

Table 2 presents the perpetual litmus test suite, which includes 34 perpetual litmus tests generated for the x86-TSO memory model. The test suite is split into two groups of tests, based on whether their target outcomes are allowed or forbidden by the specification of the x86-TSO memory model, as determined using the herd memory model simulator [18].

6 Evaluation Methodology

6.1 Testing Environment & Tools

We evaluate PerpLE using the perpetual litmus test suite from Table 2. Experiments are run on a CentOS 7.6 Linux cluster with 32 Intel Xeon E5-2667 CPUs with two threads per core. As per Section 2.1.2, the memory consistency model of CPUs in this cluster is a TSO variant [14], so we expect to only observe target outcomes from the ”Allowed” group of tests in Table 2.

We compare the performance of PerpLE to that of litmus7 in each available thread synchronization mode [9]: 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 (if available) for synchronization; and `none`, where no thread synchronization is used [8].

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.

6.2 Metrics of Interest

6.2.1 Target Outcome Occurrences

Because perpetual outcomes are determined per frame and the number of frames is polynomial in the number of iterations (N^T), 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 when using our tool. 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.

6.2.2 Heuristic Outcome Counter Accuracy

In order to evaluate our heuristic outcome counter, we determine its accuracy for the target outcome of each of the tests in our suite. In particular, for the target outcome of each test, we run both 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).

6.2.3 Testing Runtime

Since per-iteration thread synchronization dominates runtime for litmus7 in `user` mode, its removal should significantly reduce test runtime. However, the exhaustive outcome counter must examine all N^T frames for perpetual outcomes of interest (for N iterations and T 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, which only examines N frames, should preserve a considerable speedup over litmus7.

6.2.4 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 projected 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. Note that this is the most informative metric, indicating our ability to observe outcomes of interest for a given amount of available testing time.

6.2.5 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, enabling 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 4.2. 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 between n and m is exactly the skew between threads t and s around the time of iteration n in thread t.

6.2.6 Outcome Variety

One goal of perpetual litmus tests is to increase the effectiveness of memory consistency testing by enabling more thread interaction and exposing a greater variety of test outcomes. To evaluate PerpLE's performance with respect to this goal, we compare how frequently each possible outcome occurs in PerpLE and in litmus7 for the same test and number of iterations, for three indicative tests.

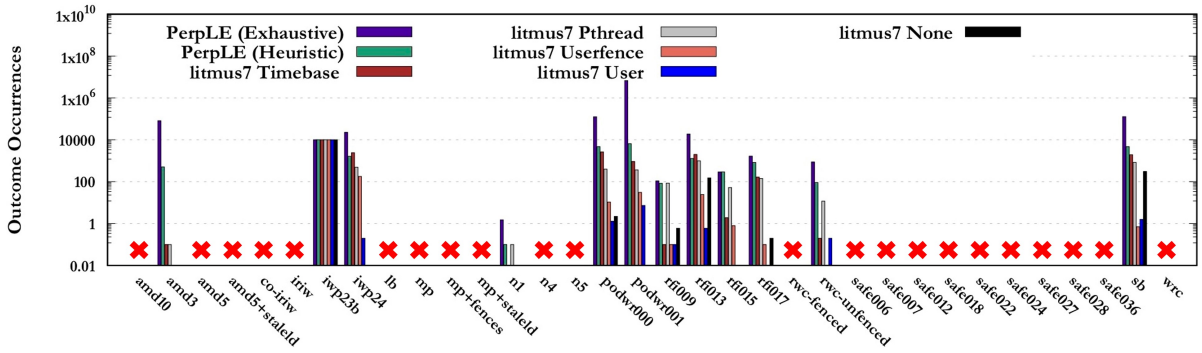

Figure 9: Target outcome occurrences for each test of the perpetual litmus suite for 10k iterations. Higher is better. PerpLE with either outcome counter generally outperforms litmus7 in most synchronization modes. A red X symbol marks each test with a target outcome that is forbidden under x86-TSO. 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.

7 Evaluation

This section evaluates PerpLE using the metrics described in Section 6. 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.

7.1 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 the exhaustive outcome counter performs strictly better than litmus7 in all cases, observing many occurrences of each target outcome. PerpLE with the heuristic outcome counter performs better than litmus7 in most cases, with the exceptions of the iwp24 and rfi013 tests, where litmus7 in the `timebase` synchronization mode marginally outperforms our tool. 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 2, many of the tests in the perpetual test suite have target outcomes that are forbidden under x86-TSO; 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

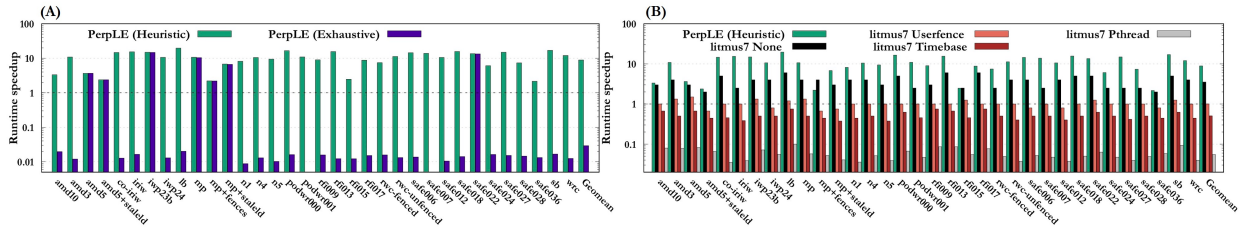

Figure 10: Runtime speedup over litmus7 in `user` mode of PerpLE using the exhaustive and heuristic outcome counter as well as litmus7 in the other synchronization modes, for each test in the perpetual litmus suite. Runtimes include both test execution and outcome counting. Higher is better.

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 this sense. 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 (see amd3, iwp24, n1, podwr001, rfi015, rfi017, rwc-unfenced).

7.2 Heuristic Outcome Counter Accuracy

Figure 9 also 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 observed.

7.3 Testing Runtime

Figure 10 presents the runtime speedup, compared to 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(A) focuses on the comparison between PerpLE when using the exhaustive or the heuristic outcome counter. As discussed previously, the runtime of the 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 such threads, so examining a quadratic number of frames makes PerpLE with the exhaustive outcome counter significantly slower than PerpLE with the heuristic outcome counter in these cases. The perpetual litmus tests where the exhaustive outcome counter performs comparably to the heuristic outcome counter, e.g. mp, only have a single

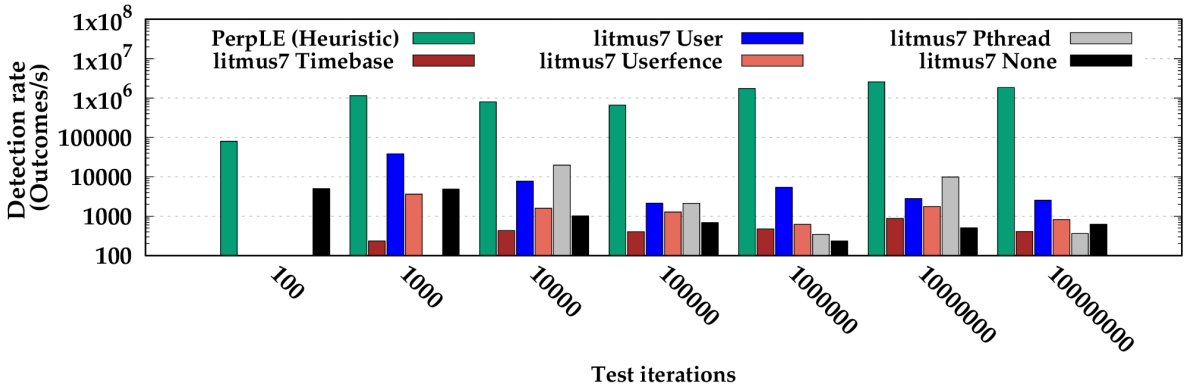

Figure 11: Target outcome detection rate of PerpLE using the heuristic outcome counter and litmus7 in different synchronization modes, for different numbers of test iterations. Bars represent the Geomean target outcome detection rate across all perpetual litmus tests with target outcomes allowed by x86-TSO. Higher is better. Note that (i) PerpLE outperforms litmus7 in most synchronization modes over two orders of magnitude and (ii) unlike PerpLE, litmus7 in most synchronization modes does not observe the target outcome for low iteration counts.

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 becomes cubic, yielding a dramatic slowdown. Across all tests, the geometric average speedup of the heuristic outcome counter over the exhaustive outcome counter is 305x.

Therefore, as expected, the exhaustive outcome counter scales poorly as the number of iterations increases, compared to the linear heuristic outcome counter. As such, the remaining evaluation will focus on the heuristic outcome counter. For the remainder of Section 7, we will use "PerpLE" to refer to PerpLE using the heuristic outcome counter. As Figure 10(B) shows 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 is comparable to that of `none`, since both use no synchronization and examine a linear number of frames. However, PerpLE offers a significantly better target outcome detection rate, as presented in Section 7.4.

7.4 Target Outcome Detection Rate

Figure 11 compares the target outcome detection rate of PerpLE and litmus7 in different synchronization modes, across an increasing number of test iterations. Each bar corresponds to the geometric mean of all tests of the perpetual litmus suite that have target outcomes allowed in x86.

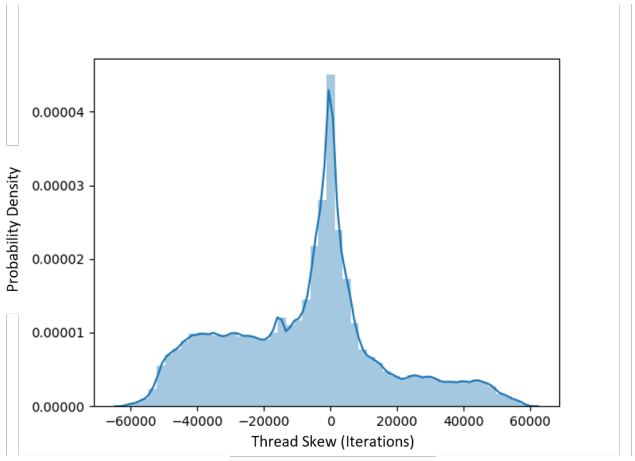

Figure 12: Probability density of the thread skew (in iterations) between the two test threads of the store buffering (sb) perpetual litmus test for 100k iterations. Store buffering (sb) is used as an illustrative example; other tests exhibit similar thread skew results based on our experiments.

PerpLE demonstrates the ability to observe target outcomes at lower iteration counts than litmus7. 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 outcome detection rate improvement is between 40.17x (over `pthread`) and 1834x (over `timebase`). PerpLE is able to scale gracefully, maintaining a high target outcome detection rate improvement for high iteration counts: between 258.79x-5074.86x for 10M and 727.28x-5074.18x for 100M iterations. Overall, the target outcome detection rate of PerpLE is typically between two and three orders of magnitude higher than that of litmus7 in any synchronization mode for most iteration counts.

7.5 Thread Skew

Figure 12 presents the probability density function of thread skew for the perpetual sb litmus test. Thread skew is a result of numerous system factors, like operating system scheduling and small differences in thread launch timing. The width of the distribution confirms that threads run far behind or ahead of each other during execution. This 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 are between operations of the same iteration (no skew). Still, the distribution is denser around 0, since system factors may delay either test thread during execution and these effects cancel out.

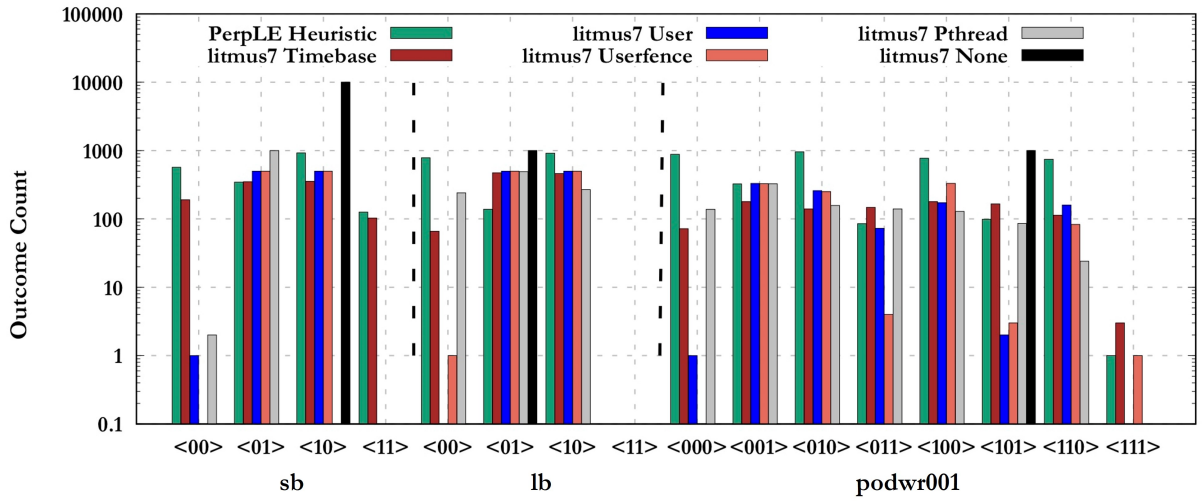

Figure 13: PerpLE using the heuristic outcome counter and litmus7 in different synchronization modes compared in terms of outcome variety for the sb, lb and podwr001 litmus tests, for 1k iterations. PerpLE samples 1k frames *per outcome*. Higher outcome variety is better. Note that the 11 outcome for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

7.6 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 were presented in full in Figure 2. Litmus7 across different synchronization modes and PerpLE 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 allowed 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 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 is typically higher than litmus7 in any synchronization mode.

As such, 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.

8 Related Work

Memory models range widely, from sequential consistency (SC) [11] to weak memory consistency models in modern architectures [19][20][21][22][23]. Several previous works have focused on formalizing the memory consistency models of different architectures, e.g. x86 [14], POWER or more generic models[18][24][25], while others have focused on language level formalization of memory models [26][27][28]. Frequently, such efforts resort to running litmus tests on hardware and comparing their empirical observations to the published memory model specifications [29][30]. 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 [31][32][33]. 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. Another line of work has focused on specifying and verifying microarchitectural implementations of consistency models using happens-before graphs [6][34][35][36][37]. This set of tools focuses on design time verification whereas PerpLE performs runtime evaluation of litmus tests against the hardware specification.

Based on formal memory consistency models, some past 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 [32][33][38][39]. The Converter tool in PerpLE extends such tools, by enabling the conversion of newly generated litmus tests to their perpetual counterparts, hence 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 [40][41][42]. Central among these is the diy suite of tools, which includes the litmus7 tool that we use in our evaluation [8]. PerpLE adds to such tool development efforts, by providing a critical twist (removal of per-iteration synchronization) on an approach users are already familiar with (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 [5] 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.

9 Conclusions

Given parallelism's centrality in computing today, memory consistency testing is critical to ensure that our 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 the need for 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 outcome 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 work focused on x86, but our approach can be applied to architectures implementing weaker memory models as well. These improvements can help expand the applicability and effectiveness of empirical memory consistency testing techniques.

Part B

Formally Verifying the DCP Subsystem of the DECADES Intelligent Storage (IS) Tiles

10 Introduction

10.1 Overview of the DECADES Architecture

The DECADES chip is being co-developed by research groups at Princeton University and Columbia University, as part of the DARPA Software-Defined Hardware (SDH) program, which aims to "build runtime-reconfigurable hardware and software that enables near-ASIC performance without sacrificing programmability for data-intensive algorithms" [43].

In particular, the DECADES chip aims to accelerate data-intensive machine learning and graph applications. It is based on a heterogeneous tiled platform architecture that uses three types of tiles: specialized accelerator tiles, processor tiles and IS tiles. Tiles communicate through reconfigurable networks-on-chip (NOCs) and all have direct access to memory. Accelerator tiles are used for highly specialized but frequent types of computation, like convolution and matrix multiplication, while processor tiles are used for all remaining computation, as well as for general-purpose computing. IS tiles are pivotal for facilitating large-scale on-chip data movement, which is often required by the types of applications targeted by DECADES.

10.2 The DCP Subsystem

The focus of this project is verifying the Decoupled Consumer-Producer (DCP) subsystem [44] within the IS tiles. This subsystem allows IS tiles to act as a tile-to-tile communication buffer within a parallel Decoupled-Access Execute (DAE) architecture. A DAE architecture provides a way to perform memory accesses and computation in parallel without relying on speculation [45][46][47]. Code is sliced into two threads, with the *access* thread handling memory access and address computation, while the *execute* thread performs other computation. Ideally, the access

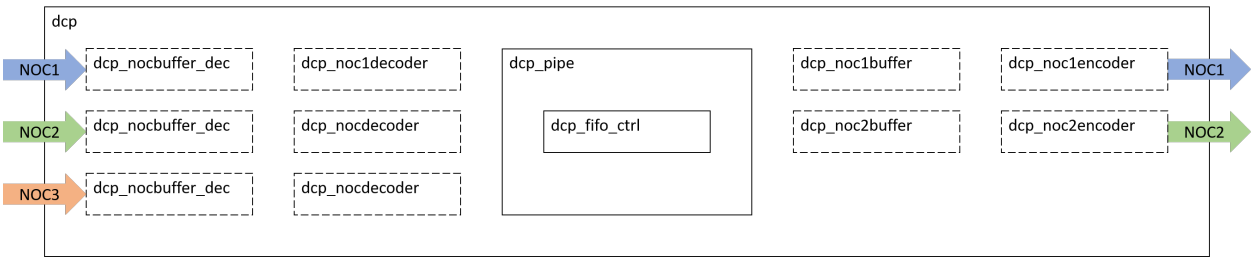

Figure 14: The DCP subsystem, organized into its constituent Verilog modules. Solid outlines indicate modules that were actively verified at a unit level for this project.

thread runs ahead and procures data, which is buffered in a FIFO until the execute uses it for complex value computation. This approach is especially useful for the parallel processing of kernels bound by memory latency. The DCP subsystem handles the data buffering stage of the process within the DECADES architecture.

As presented in Figure 14, the DCP subsystem includes several Verilog modules; `dcp`, `dcp_pipe` and `dcp_fifo_ctrl` are new designs, while the remaining modules are communication buffers and NOC encoders/decoders borrowed from existing work [48]. In the interest of efficiency, only *new* modules were actively verified at a unit level for this project. Any potential bugs in the older modules that were missed during earlier verification would be evident during the verification of the top-level `dcp` module. Case study 1 in Section 16.2 presents one such bug, related to the integration of the `dcp_noc1buffer` module. Briefly, the functionality of the three actively verified modules is as follows:

- The `dcp_fifo_ctrl` module manages a scratchpad, parts of which can serve as FIFOs, each between an access and an execute core. These FIFOs hold data returned from memory at an access core's request until the corresponding execute core consumes it.
- The `dcp_pipe` module processes incoming and outgoing requests and responses from other tiles and/or memory. It is responsible for sending data requests to memory, handling incoming data responses from memory and forwarding data from a FIFO to the appropriate execute core as needed.
- The `dcp` module provides the interface between the DCP subsystem and the Networks-on-Chip (NOCs), by adding buffering capability for incoming and outgoing messages, as well as encoders and decoders for messages based on the NOC specification.

11 Background

11.1 Verification Using Simulation

Verification is a critical part of the development of programming systems, especially when the scale of the system is large and/or multiple parties are involved in the development. Most frequently, such verification takes the form of *simulation*, i.e. it is centered around *tests*, individually generated by the development team to exercise a system.

If tests are generated manually, valuable programmer time is wasted, while increasing the risk of introducing new bugs *in the testing code* itself. The resulting tests are also more likely to concern "obvious" use cases (normal use or easy-to-imagine corner cases), which are less likely to have been overlooked by the designer when developing the system in the first place. To avoid these issues, test generation can be automated instead, using a process like the Universal Verification Methodology (UVM) [49]. In UVM, constraints are placed on the values of different system inputs and random tests are then generated within these constraints.

However, in a large complex system, automatically generated tests may still only end up exercising a small subset of the total space of system behaviors, with potentially critical bugs remaining undetected. As such, it is difficult to use simulation-based testing as a basis for formal claims about design correctness.

11.2 Formal Verification: From Tests to Properties

Given the above concerns, a second type of verification is more suitable when formal correctness guarantees are required: formal verification (FV). As defined by Seligman et al., FV is the use of tools that mathematically analyze the space of possible behaviors of a design, rather than computing results for particular values [50]. That is, there is a shift from an emphasis on *tests*, where the system response to each test is examined separately, to an emphasis on *properties*. Instead of attempting to cover the complete space of possible system behaviors with a large number of individual tests, mathematical tools are employed to prove a set of properties guaranteeing the correctness of the system.

FV accelerates verification set-up, since defining a property is faster and less error-prone than developing the equivalent simulation testbench. FV also provides confidence about the verification results. With simulation, it is hard to prove the exhaustiveness of a set of test cases in order to proclaim a system's correctness. With FV, a more formal picture of the verification coverage emerges instead, which makes it possible to quantify confidence in the correctness of a system.

However, in exchange for these formal guarantees, FV can be slower to actually run than simulation. For each property, FV proceeds by rounds of increasing *depth*. At a depth of x clock cycles, the FV tool examines all possible *traces* of length x, which are combinations of sequences of values for all possible inputs, in search for violations of the property being verified. This creates a large combinatorial search space as the depth increases, slowing down FV.

On the upside, this means that FV tools can provide bounded proofs as they run: if a property is currently being verified at depth x, no violating trace of up to $x-1$ clock cycles exists. Still, FV is most efficient for smaller-scale designs, or for unit testing parts of a larger design, which is the focus of this project. In order to fully test a larger design, a mixed approach would be better, combining unit-level FV with system-level simulation. The properties used for unit-level FV can also serve as assertions during simulation, reducing the additional set-up required.

12 Methodology

12.1 General Approach

12.1.1 Bottom-Up Verification

The three modules actively verified for this project have a hierarchical relationship: `dcp_fifo_ctrl` is a sub-module within `dcp_pipe`, which is itself a sub-module within `dcp`. This relationship provides the opportunity to structure the verification in a "bottom-up" manner, consisting of three phases, as presented in Figure 15. In Phase I, all assertions concern the end-to-end behavior of just the `dcp_fifo_ctrl` module. If the FV tool fails to prove an assertion by finding a counter-example that violates guarantees provided by the specifications of the `dcp_pipe` and `dcp` modules, or by any non-DCP components, assumptions are introduced for these guarantees as needed, in order to focus on verifying just `dcp_fifo_ctrl`.

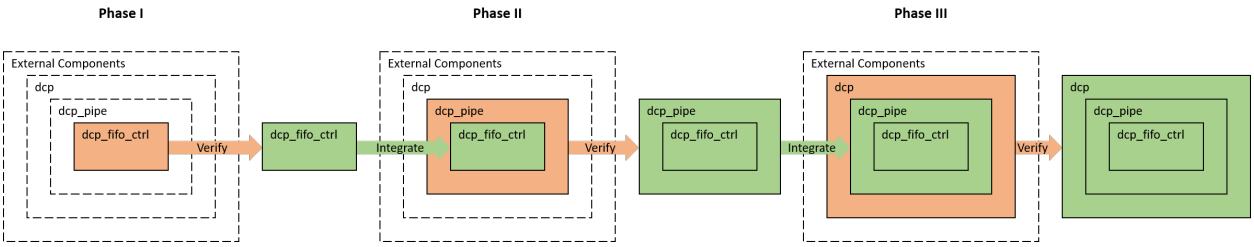

Figure 15: Flow of our bottom-up verification procedure. Light orange modules are being verified, while green modules have been verified already. In each phase, whenever necessary, the behavior of components with dashed outlines can be modeled by writing SVA assumptions based on their specifications.

Once the `dcp_fifo_ctrl` module has been verified, we move to Phase II. There, the `dcp_fifo_ctrl` module is integrated into the `dcp_pipe` module, with new assertions all concerning the end-to-end behavior of the `dcp_pipe` module. Any assumptions made in Phase I are also turned into assertions. However, if needed, assumptions are again introduced for any guarantees provided by the specification of the `dcp` module, as well as by any non-DCP components.

Finally, once the `dcp_pipe` module has also been verified, we move to Phase III. There, the `dcp_fifo_ctrl` and `dcp_pipe` modules are integrated into the `dcp` module, with new assertions all concerning the end-to-end behavior of the top-level `dcp` module. Any assumptions made in Phase II are also turned into assertions. However, if needed, assumptions are again introduced for any guarantees provided by any non-DCP components.

12.1.2 Design and Verification Interaction

Since verification aims to ensure that a system conforms to its specification, thoroughly documenting this specification should be the very first step of any verification project, so as to provide a concrete basis for the designer-verification engineer interaction. This is even more critical in FV, where the lack of a detailed low-level specification can result in incorrect assumptions/assertions, causing the FV tools to explore unrealistic system states and report "false positive" assertion violations. However, keeping detailed documentation of the specification up-to-date while still developing the design necessarily hampers productivity. Even worse, since technical realities may keep pushing the design in very different directions, time spent in constantly updating the documentation is most likely time wasted, because further updates might be needed again very soon.

When working with a smaller team, like that of this project, it might instead be preferable to adopt a more agile approach and extend methodologies from rapid software development to hardware development. In particular, team members can informally communicate day-to-day changes to the specification, using scrum-style updates [51], while documentation updates can be cumulative and take place less frequently. For this project in particular, we started verifying in parallel with the last stages of design work, before complete documentation of the low-level specification was available. As such, we made every effort to verify at the level of end-to-end properties, involving only inputs/outputs of our modules, for which the specification was already clear. We then used the counter-examples found by JasperGold to gradually refine our low-level specification, dynamically making decisions about certain low-level details, as illustrated by the case studies in Section 16.

12.2 Using SystemVerilog Assertions (SVA)

For this project, we use SystemVerilog Assertions (SVA) to specify the properties that we aim to verify. The SVA language is part of SystemVerilog, the IEEE-standardized version of Verilog and some of its extensions [52]. Although other options exist, like the Property Specification Language (PSL) [53], SVA has seen the widest industry adoption for verifying register transfer level (RTL) assertions [50]. After specifying properties in SVA, we use JasperGold [54], a FV tool that performs the verification and informs us of the results (proof or counterexample, for each property), alongside relevant metrics about the set of verified properties. This Section briefly presents some basic elements of SVA.

12.2.1 Vocabulary

At the most basic level, the SVA language has the following nested layers of complexity:

- Expressions: The same Boolean expressions generally found in SystemVerilog, only involving signals and Boolean operators.
- Sequences: Expressions together with timing relationships between them, such as the ##x delay operator (as defined in section 12.2.3).

- Properties: Expressions and sequences together with additional operators for more complicated concepts, such as the `|->` implication operator (as defined in section 12.2.3).
- Assertions: Properties together with one of the keywords presented in section 12.2.2, specifying how the properties should be interpreted by a FV tool.

12.2.2 Types of Assertion Statements

SVA distinguishes among three different kinds of assertions: `assert`, `assume` and `cover` [52]. From now on we will refer to these three kinds as "assertions", "assumptions" and "cover points" respectively to avoid confusion, even though all of them are simply different kinds of assertions per the IEEE standard. We will use the term "assertion statements" to refer collectively to all three kinds. Although the syntax for the three kinds is the same, as presented below, the way that an assertion statement will be subsequently interpreted by a platform like JasperGold has important differences. In particular:

- Assertions (using `assert`): Treated as proof targets, properties that we want to verify a system never violates. They typically involve internal signals or outputs of the module being verified.
- Assumptions (using `assume`): Treated as constraints on the verification environment, which accelerate FV by letting the FV tool ignore certain cases. They typically involve inputs of the module being verified.
- Cover points (using `cover`): Treated as "must-consider" cases, which we want to make sure are observed during FV. They can involve inputs or outputs of the module being verified.

```
// Assert that a counter remains below 10.
as__cnt_10: assert property (count < 10);

// Assume that all operations are either loads or stores.
am__ld_or_st: assume property ((op ==‘OP_LOAD) || (op == ‘OP_STORE));

// Cover the case where two valid requests arrive at once.
co__sim_requests: cover property (req1_val && req2_val);
```

Note that FV should initially be attempted without any assumptions, which implies that no cover points are necessary either, since FV tools will consider all cases by default. However, if assertions are failing because of counter-examples corresponding to unrealistic cases, such as cases that some other system component guarantees will not occur, assumptions can be gradually introduced to rule out these spurious assertion violations. As assumptions are introduced, there is a growing risk of over-constraining the state space exploration performed by FV tools, which can be revealed by using appropriate cover points.

12.2.3 Timing

We now introduce some SVA features used to address the passage of time, both in specifying and in evaluating properties. The first feature is the difference between *immediate* and *concurrent* assertions. Immediate assertions behave like assertions in "traditional" languages like C or Java, in the sense that they are only checked whenever they are visited in procedural code. However, this necessarily makes them unable to check properties involving the passage of time. As such, while we can still use them for e.g. argument safety checking, they are of limited use in our verification context. On the contrary, concurrent assertions are checked according to some clock. This both limits spurious assertion failures due to transient system state between clock edges and allows us to specify and check properties across time. Therefore, we only use concurrent assertions in this project. Concurrent assertions are signified by the keyword `property` after the keyword `assert` [50]. The second feature is the existence of timing-related operators and functions, some of which we describe below:

- The ##*x* operator: As part of a sequence, this operator indicates a delay of *x* clock cycles between two events. For example, `(sig_1 == 2'b01) ##1 (sig_2 == 2'b11)` will be evaluated by checking the value of `sig_2` one cycle later than the value of `sig_1`.
- The `|->` and `|=>` operators: We use them in triggered implication constructs, where they are preceded by some sequence and followed by some property. In this type of construct, the property after the operator is only checked when the sequence before the operator is matched. The `|->` operator checks the property in the same cycle that the sequence was matched, while the `|=>` operator checks the property one cycle later.

- The `s_eventually()` operator: The `s_eventually(`*expr*`)` operator indicates that *expr* must eventually be true. The `s_` prefix here stands for "strong", indicating that even in an infinite trace, the expression must be true at some point.
- The `$past()` and `$stable()` functions: The `$past(`*expr*`)` function returns the value of the expression *expr* one clock cycle earlier. The `$stable(`*expr*`)` function returns 1 iff the value of the expression *expr* did not change between the previous and the current clock cycle.

12.3 Verification Techniques

This Section will present two common verification techniques used across all three phases of this project. More details about how exactly these techniques were employed can be found in Appendices A-C, where the SVA code for all verified properties is presented.

12.3.1 Behavioral Models

Frequently, properties that involve multiple different events across different clock cycles must be verified. In such cases, the expressive power of sequences and expressions alone might not be sufficient to state the desired properties elegantly. A better approach is for the verification engineer to use the system specification to create a behavioral model for a part of the system. The desired properties can then be expressed as comparisons between the state of the model and the state of the real system.

The effectiveness of this approach is based on two key observations, the validity of which can be verified with the help of metrics provided by the FV tool (e.g. code coverage). First, that it is unlikely for the system designer and the verification engineer to both create the same bug when independently implementing some part of the system, especially if the former writes synthesizable code while the latter writes a simpler behavioral model. Therefore, the case of false "proofs" is largely ruled out: when model and system agree, they are much more likely to be both right than both wrong. Second, that the behavioral model built by the verification engineer is less bug-prone than the real design due to its simplicity. Thus, in cases where model and system disagree, any bugs in the model can be easily spotted and eliminated, with the FV tool only revealing possible bugs in the real design thereafter.

As an example, the model used for verifying property II-7 is presented below, which keeps track of which FIFOs have an access core configured at any time, as per Section 13.1.3.

```
//// Model for II-7.

// Keep track of FIFOs with an access core configured.
reg [`FC_FIFO_SIZE-1:0] is_aconfigured_r;

// Model state update logic.
always_ff @(posedge clk) begin
    if (!rst_n || invalidate) begin
        is_aconfigured_r <= {`FC_FIFO_SIZE{1'b0}};
    end
    else begin
        if (adeconfiguring) begin
            is_aconfigured_r[fifo_idx] <= 1'b0;
        end
        if (aconfiguring && !invalidate) begin
            is_aconfigured_r[fifo_idx] <= 1'b1;
        end
    end
end

////

// II-7: Verify which FIFOs have an access core configured at any time using a
// model.
as__aconf_model_match: assert property(is_aconfigured_r[symb_fifo] ==
    c0_aconfigured_r[symb_fifo]);
```

12.3.2 Symbolic Variables and Generate-For Loops

Often, properties may concern arbitrary or multiple instances of identical constructs; for example, in the context of this project, properties concerning any one scratchpad entry or all FIFOs. In such cases, it is clearly inefficient to explicitly write the same property multiple times, simply tweaking the instance it refers to. Instead, the SVA language provides two alternatives: symbolic variables, which can act as an *existential* qualifier across all instances, and generate-for loops, which can act as a *universal* qualifier across all instances.

Symbolic variables have no value explicitly assigned to them. When used in assertions, the FV tool will treat them as "free inputs" and check if there exists a possible sequence of values for them which leads to an assertion violation. As an example, a symbolic variable can be used to index into the array of scratchpad entries, as in property I-1 presented below. This property checks that an entry that is not currently in use cannot be valid in the next clock cycle (for reasons explained in Section 13.1). In order for the FV tool to prove I-1 as written, it must fail to find a counter-example for *any* value of the symbolic variable, thus proving the property for *each* scratchpad entry, more efficiently than having to prove a separate property for each entry.

Symbolic variables are often accompanied by assumptions to guide the FV tool. In this project, one assumption always used for symbolic variables is that of stability over time. For example, without assuming that `symb_idx` is stable, the property I-1 as written below would instead be interpreted as "if *any* entry is not currently in use, *no* entry can be valid in the next clock cycle". One easy way for the FV tool to find a spurious "counter-example" here would be to simply pick different values of `symb_idx` for the two clock cycles. By including the assumption, the FV tool is still free to pick any entry, but it is constrained to thereafter only look at the *same* entry over time.

```
wire [FC_GLOBL_IDX-1:0] symb_idx = 'x;
am__symb_idx_stable: assume property($stable(symb_idx));

// I-1: An entry that is not currently in use cannot be valid in the next clock
// cycle (it must first get reserved and then get filled).
as__st_legal_out_of_inv: assert property (!data_use[symb_idx] |=>
    !data_val[symb_idx]);
```

However, for all their power, symbolic variables are not suitable when writing assumptions instead. Since assumptions are meant as constraints, *all of them* must be enforced simultaneously - not just one arbitrary constraint each time, as dictated by the value that the FV tool chose for the symbolic variable. Therefore, we instead use generate-for loops, which create an instance of the loop body in each iteration. As an example, consider property II-17 presented below, which assumes that each core can have up to one outstanding request to the IS tile. Written using symbolic variables, the assumption would instead be interpreted as "*some* core can have up to one outstanding request to the IS tile", which is clearly much weaker. Written using a generate-for loop instead, an instance of the assumption would be created *for each core*.

```
// A symbolic core ID
wire [`PACKET_HOME_ID_WIDTH-1:0] symb_core = 'x;
am__symb_core : assume property ($stable(symb_core));

// Keep track of the outstanding requests to the IS tile per core
logic [NUM_CORES-1:0][1:0] core_outstanding_req_r;

// [Model update logic omitted, see Appendix B]

// II-17: Each core can have up to one outstanding request to the IS tile.
// Incorrect assumption using symbolic variables.
am__core_max_1_outstanding: assume property(core_outstanding_req_r[symb_core] <
    2'b10);

// II-17: Each core can have up to one outstanding request to the IS tile.
// Correct assumption using generate-for loop.
generate for (genvar i = 0; i < NUM_CORES; i = i + 1) begin : max_1_assume
    am__core_max_1_outstanding: assume property(core_outstanding_req_r[i] <
        2'b10);
end endgenerate
```

13 Phase I: Verification at the `dcp_fifo_ctrl` Level

13.1 Specification of the `dcp_fifo_ctrl` Module

The `dcp_fifo_ctrl` module is responsible for managing a fixed-size scratchpad of N 4-byte entries. These scratchpad entries can be used to form at most F FIFOs. For the DECADES chip, technical constraints related to other parts of the system dictate the values $N = 256$ and $F = 8$.

The motivation for allowing multiple FIFOs on the same IS tile comes from the case of "asymmetric" decoupling within a single application, either because of multi-threading or because of performance imbalances. In the latter case, there can be a significant difference between the rate with which an access core issues data requests and the rate with which an execute core consumes the data; thus, deviating from the 1:1 ratio of access to execute cores could help limit stalling. For example, for a given application and system load, if an access core issues 2 data requests per unit time, but the corresponding execute core can only consume 1 piece of data per unit time, it would make sense to use two execute cores (`ex1` and `ex2`) per access core (`ac`), taking turns between them. This requires two lines of communication: one where `ac` requests data on behalf of `ex1`, and another where `ac` requests data on behalf of `ex2`. With the multi-FIFO design, both of these can be housed in a single IS tile, increasing tile utilization and decreasing the number of IS tiles that the chip needs for a given number of processor tiles.

13.1.1 Scratchpad Entry States

As shown in Figure 16, each of the N 4-byte scratchpad entries can be in one of 4 states We use the term " in use" to describe an entry in the `PRI`, `SEC` or `VAL` state, and the term "valid" to describe an entry in the `VAL` state. The 4 states are briefly described below, while the transitions among them are explained in Sections 13.1.4 and 13.1.5.

- The `INV` state: The entry does not contain meaningful data and is not currently involved in any outstanding memory requests. On reset, all entries are in the `INV` state.
- The `PRI` state: The entry does not currently contain meaningful data, but is involved in an outstanding memory request; that request either expects a 4-byte response, or expects an 8-byte response of which the *4 least significant bytes* will be stored in the current entry.

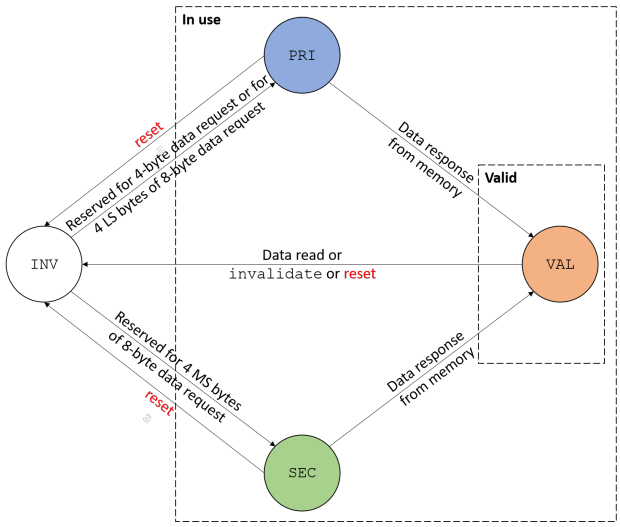

Figure 16: Finite-state machine for the 4 states of a scratchpad entry.

- The `SEC` state: The entry does not currently contain meaningful data, but is involved in an outstanding memory request; that request expects an 8-byte response of which the *4 most significant bytes* will be stored in the current entry.
- The `VAL` state: The entry currently contains meaningful data, that could be provided to the appropriate execute core upon request.

Note that even for 4-byte requests, memory will respond with 8 bytes of data, the 4 most significant of which will be meaningless. The data response itself will give no indication of when that is the case. As such, differentiating between the `PRI` and `SEC` states is essential, in order keep track of the size of an expected data response and treat the 4 most significant bytes appropriately. Case study 2 in Section 16.3 further elaborates on this issue.

13.1.2 Valid and Invalid FIFOs

Each of the F possible FIFOs is indexed by a number between 0 and $(F-1)$ and can be either valid or invalid. A valid FIFO i has a specified `top` pointer and contains the scratchpad entries $\texttt{top}_{i-1}$ to $\texttt{top}_i - 1$, inclusive (let $\texttt{top}_{-1} = 0$ for notational simplicity). A valid FIFO may be *empty*, when none of its entries are in use, *full*, when all of its entries are in use, or neither.

After a system reset, all FIFOs are invalid. An invalid FIFO can become valid by being *created* by a core. A core can create a new FIFO by sending an appropriate configuration message to the IS tile, specifying the desired size of the FIFO (in 4-byte entries). The minimum size for a FIFO is 8 entries and the maximum size is $\frac{N}{2}$ entries. If $q < F$ is the number of currently valid FIFOs on this tile, and there are at least as many entries above $\texttt{top}_{q-1}$ (inclusive) as requested, FIFO q will become valid and its `top` pointer will be set according to its size. There is no way to subsequently change $\texttt{top}_i$ while FIFO i remains valid. The index of the newly-created FIFO is then returned to the core that requested its creation, for future reference.

A FIFO can return to the invalid state in three ways. First, when the entire system (i.e. all tiles as well as memory) is reset. All scratchpad entries are then immediately reset to `INV` and all FIFOs are immediately reset to invalid. Second, when the execute core (see Section 13.1.3) of a FIFO de-configures itself, it can also choose to render that FIFO invalid. In this second case, no new FIFOs can be created on the tile until all currently valid FIFOs have become invalid, to avoid internal fragmentation of the scratchpad. Since the intended use case is for same-tile FIFOs to be used by the same application, we expect all FIFOs to be created temporally close anyway, when the application is launched. As such, temporarily not being able to create additional FIFOs well into the execution of the application is unlikely to create issues. Third, the tile may receive an `invalidate` signal, which is meant as an error recovery feature. In this case, the tile will wait for any outstanding memory requests and then reset all scratchpad entries to `INV` and all FIFOs to invalid. The reason why this wait is essential is illustrated by case study 3 in Section 16.4. In this third case as well, no new FIFOs can be created on the tile until all currently valid FIFOs have become invalid.

Each valid FIFO i also has a head_i and a tail_i pointer, both of which are initialized to $\texttt{top}_{i-1}$ upon FIFO creation. When incremented, each of these pointers wraps around to only point to entries within its own FIFO. For example, each of the pointers corresponding to FIFO i will only point from entry $\texttt{top}_{i-1}$ to entry $\texttt{top}_i - 1$, inclusive. This wraparound may be omitted for simplicity of presentation in later Sections, but it is a technical reality of the design.

13.1.3 Configuration of Access and Execute Cores

Each valid FIFO may have a configured access core and a configured execute core. Upon creation, the configured access core of a FIFO becomes the core that issued the creation request, while by default no core is configured as the execute core for that FIFO. As long as a FIFO does not have an access (or execute) core configured, any core can request to configure itself as that FIFO's access (or execute) core, which will be successful. The access (or execute) core for a FIFO can also request to de-configure itself, which will be successful. Specifically when the execute core for a FIFO is de-configuring itself, it can also choose to make that FIFO invalid, as per Section 13.1.2, which will also automatically de-configure the access core for that FIFO.

Both after a system reset and upon receiving the `invalidate` signal, the access and the execute cores of all FIFOs are immediately de-configured. Additionally, in the case of the `invalidate` signal, no core is allowed to configure itself as an access or execute core for a FIFO on this tile until the invalidation process is complete, as per Section 13.1.2.

13.1.4 Loading Data from a FIFO

The execute core of a valid FIFO i can load from it either 4-byte or 8-byte pieces of data. This is only successful if the entries pointed to by head_i, as well as by $\text{head}_i + 1$ (with wraparound) if attempting to load 8 bytes, are currently in the `VAL` state; the load request will stall until this condition is met. Once the load is successful, the data in this entry, or these entries if attempting to load 8 bytes, is returned to the execute core (we call this *reading* these entries). The head_i pointer is then incremented by 1 or 2 (with wraparound) depending on the size of the loaded piece of data. Any entries that were read transition from the `VAL` to the `INV` state.

13.1.5 Storing Data into a FIFO

The access core of a valid FIFO i can store into it either 4-byte or 8-byte pieces of data, following a two-step process inside the STORE pipeline, further described in Section 14.1.3. First, the appropriate number of consecutive entries (1 or 2) is *reserved* at tail_i. This is only successful if the entries pointed to by tail_i, as well as by $\text{tail}_i + 1$ (with wraparound) if attempting to store 8 bytes, are currently in the `INV` state; the store request will stall in this step until this condition is

met. Once this first step is successful, the state of the entry pointed to by tail_i is changed to PRI and, if attempting to store 8 bytes, the state of the entry pointed to by $\text{tail}_i + 1$ (with wraparound) is changed to SEC. The tail_i pointer is then incremented by 1 or 2 depending on how many entries were reserved.

Second, as soon as the requested piece of data arrives from memory, the appropriate reserved entries will be *filled* with the data and their state will be updated to VAL. Procuring the data may require from one to several additional clock cycles from when the entries were reserved, depending on the origin of the data within the memory hierarchy.

13.2 Relevant Properties

This Section presents a high-level description of all the properties verified during Phase I, to ensure that the dcp_fifo_ctrl module abides by the specification presented in Section 13.1. The corresponding SVA code for each of these properties is available in Appendix A. In Section 13.2.2, we also present the assumptions that were needed during Phase I (I-13 and beyond) regarding the correct functionality of modules outside dcp_fifo_ctrl. These assumptions were changed to assertions for Phases II and III.

13.2.1 Phase I Assertions

The properties below verify that FIFOs are managed by dcp_fifo_ctrl according to the specification. Properties I-1 to I-6 concern the states of individual scratchpad entries, as well as the transitions between states (see Figure 16). Properties I-7 to I-11 concern the state of entire FIFOs, as well as the impact of that state on the legality of different operations. Property I-12 ensures that any data read from a scratchpad entry matches the data with which that entry had been filled.

- I-1: An entry that is not currently in use cannot be valid in the next clock cycle (it must first get reserved and then get filled).
- I-2: An entry in the PRI or SEC state cannot directly transition to the INV state (it must first be filled and then be read), unless we reset the system.
- I-3: A valid entry is also in use (bookkeeping correctness check).
- I-4: A valid entry can never be in PRI or SEC in the next clock cycle.

- I-5: A valid entry stays valid if we don't read it, reset the system or use `invalidate`.
- I-6: A valid entry is no longer valid once we read it.
- I-7: We only confirm entry reservation requests for non-full FIFOs.
- I-8: If we reject an entry reservation request for a valid FIFO, either the FIFO was full, or 2 entries where requested while only 1 was not in use.
- I-9: We only successfully load data from non-empty FIFOs.
- I-10: When storing data, the tail is incremented by the correct amount.
- I-11: When loading data, the head is incremented by the correct amount.
- I-12: Any data loaded from a scratchpad entry is the same data that was stored earlier.

13.2.2 Phase I Assumptions

The properties below are assumptions that were needed during Phase I to ensure appropriate inputs to the `dcp_fifo_ctrl` module. They concern validity checks for different operations that should be performed by `dcp_pipe` on behalf of `dcp_fifo_ctrl`.

- I-13: Any valid incoming fill data is only intended for entries in use, but not valid.
- I-14: Any valid FIFO creation requests concern FIFOs that were not already valid and have ensured that enough scratchpad entries are available.
- I-15: We only receive entry reservation requests for valid FIFOs.
- I-16: We only receive load requests for valid FIFOs.

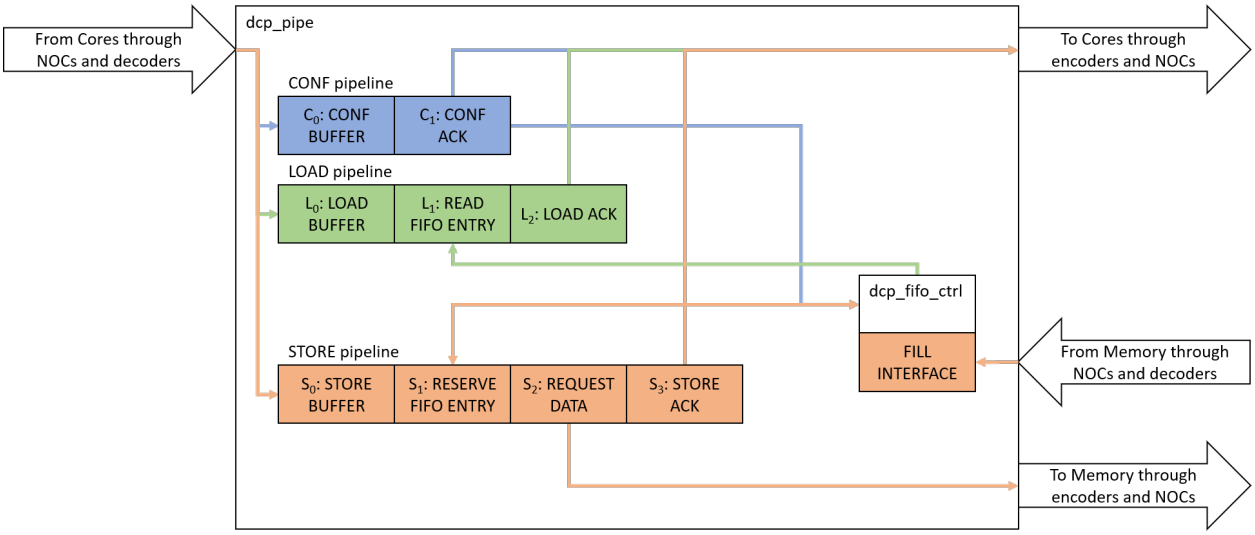

Figure 17: Internal diagram of the `dcp_pipe` module, showing the three pipelines and the flow of information. Figure adapted from [44].

14 Phase II: Verification at the `dcp_pipe` Level

14.1 Specification of the `dcp_pipe` Module

The `dcp_pipe` module is tasked with managing a `dcp_fifo_ctrl` module, by interpreting decoded incoming messages and forwarding outgoing messages to the NOC encoders. As shown in Figure 17, it uses three distinct pipelines: the CONF pipeline, intended for configuration and illegal messages; the LOAD pipeline, intended for requests to load data from a FIFO; and the STORE pipeline, intended for requests to store data into a FIFO.

14.1.1 The CONF Pipeline

The CONF pipeline is responsible for handling messages not recognized as valid load or store requests, which it performs in two stages. This includes both messages that are altogether illegal (e.g. specifying a non-existent opcode) and legitimate configuration messages. In the former case, the illegal message goes through the CONF pipeline and an ACK message is issued at stage C_1, but there is no additional effect on the state of the `dcp_pipe` or the `dcp_fifo_ctrl` modules. In the latter case, an appropriate configuration message can have a range of effects. For example, we already saw some configuration messages in Sections 13.1.2 and 13.1.3, related to FIFO creation and the configuration of access/execute cores.

Stage C_0 is responsible for determining whether an incoming message is not a valid load or store request. It also checks whether it is a well-formed configuration message, sent by a core that has the necessary permissions for that type of message. If so, the CONF pipeline sets up any necessary actions stemming from the message. Once stage C_1 is available, these actions take effect as the message moves to stage C_1 at the next clock. At that point an ACK message is also forwarded to the NOC2 buffer, directed at the source core of the configuration message. Even if the message was not legitimate, in which case it will not alter the state of DCP, it will still move to C_1 in this manner and an ACK will be forwarded. Once the NOC2 buffer has received this outgoing message, the original message exits the CONF pipeline at the next clock.

14.1.2 The LOAD Pipeline

The LOAD pipeline is responsible for handling messages recognized as *valid load requests*, which it performs in three stages. Stage L_0 is responsible determining whether an incoming message is a valid load request. If so, at the next clock the request is clocked into one of F slots in an L_0 buffer, with each slot corresponding to one of the FIFOs, according to the index of the FIFO from which the load request aims to load data. Note that the appropriate buffer slot cannot already be occupied. Only the configured execute core for some FIFO can issue a *valid* load request for data from that FIFO, so no other core can have an outstanding request occupying that slot. Moreover, each core can only have one outstanding request, so the execute core for some FIFO i cannot itself send additional valid load requests while one of them is still occupying slot i in the L_0 buffer.

In each clock cycle, if stage L_1 is available, a round-robin arbiter selects an occupied slot i of the L_0 buffer. If the request in that slot aims to load $4x$ (4 or 8) bytes, and there are at least x valid entries at head_i (with wraparound), that request is moved to stage L_1. If needed (e.g. if FIFO_i is currently empty), a load request will stall in slot i of the L_0 buffer until this condition is met and the arbiter selects i. In stage L_1, the next 1 or 2 entries of FIFO i are read. Then, head_i is incremented by the number of read entries. Once stage L_2 is available, the data from the read entries is clocked into an output register at the next clock, as the load request moves to stage L_2.

In stage L_2, the contents of the output register are forwarded to the NOC2 buffer, to be returned to the core issuing the load request as part of an ACK message. Once the NOC2 buffer has received this outgoing message, the load request exits the LOAD pipeline at the next clock.

14.1.3 The STORE Pipeline

The STORE pipeline is responsible for handling messages recognized as *valid store requests*, which it performs in four stages. Stage S_0 is responsible determining whether an incoming message is a valid store request. If so, at the next clock the request is clocked into one of F slots in an S_0 buffer, with each slot corresponding to one of the FIFOs, according to the index of the FIFO into which the store request aims to store data. As with the L_0 buffer, the appropriate S_0 buffer slot cannot already be occupied, by the same argument.

In each clock cycle, if stage S_1 is available, a round-robin arbiter selects an occupied slot i of the S_0 buffer. If the request in that slot aims to reserve $4x$ (4 or 8) bytes, and there are at least x not-in-use entries at tail_i (with wraparound), that request is moved to stage S_1. If needed (e.g. if FIFO_i is currently full), a store request will stall in slot i of the S_0 buffer until this condition is met and the arbiter selects i. In stage S_1, entries in FIFO i are *reserved*: the state of the entry at tail_i is updated to `PRI`, while the state of the next entry in FIFO_i (with wraparound) is updated to `SEC` if reserving 2 entries. Then, tail_i is incremented by the number of reserved entries. Once stage S_2 is available, the store request in stage S_1 moves to stage S_2 at the next clock.

In stage S_2, a memory data request may be forwarded to the appropriate NOC buffer. The message ID of this request encodes the FIFO and the entry that the 4 least significant bytes of the returned data should occupy, the same entry which was previously reserved in stage S_1 and had its state updated to `PRI`. Once the appropriate NOC buffer has received this data request and stage S_3 is available, the store request moves to stage S_3 at the next clock.

In stage S_3, an ACK message is forwarded to the NOC2 buffer, to be returned to the core issuing the store request. Once the NOC2 buffer has received this outgoing message, the store request exits the STORE pipeline at the next clock.

14.1.4 The Fill Interface

The fill interface is responsible for routing any data returned from the memory hierarchy to the `dcp_fifo_ctrl` module, as appropriate. Note that the memory hierarchy always returns 8 bytes of data; if we only requested 4 bytes, the 4 most significant bytes returned will be meaningless, but there will be no indication of this fact in the incoming message from memory.

The message ID of any returned data is the same as the message ID of the corresponding data request, so it encodes the FIFO and the specific entry that the 4 least significant bytes of the returned data should occupy (see Section 14.1.3). The fill interface first verifies that this information is valid: the specified FIFO should still be valid and the specified entry must be in the `PRI` state. If so, that entry is filled with the 4 least significant bytes of the returned data. Then, the state of the next entry (with wraparound) in the same FIFO is checked: if it is `SEC`, we must have been expecting 8 bytes from memory, so that entry is filled with the 4 most significant bytes returned. Any entries filled will also be updated to `VAL` by the `dcp_fifo_ctrl` module.

14.2 Relevant Properties

This Section presents a high-level description of all the properties verified during Phase II, to ensure that the `dcp_pipe` module abides by the specification presented in Section 14.1. The corresponding SVA code for each of these properties is available in Appendix B. In Section 14.2.2, we also present the assumptions that were needed during Phase II (II-17 and beyond) regarding the correct functionality of modules outside `dcp_pipe`. These assumptions were changed to assertions for Phase III.

14.2.1 Phase II Assertions

The properties below concern the functionality of individual pipelines: properties II-1 to II-9 concern the CONF pipeline; properties II-10 to II-12 concern the LOAD pipeline; and properties II-13 to II-15 concern the STORE pipeline. Property II-16 is equivalent to the logical AND of properties II-9, II-12 and II-15.

- II-1: A core can configure itself as an access core.
- II-2: A core can de-configure itself as an access core.
- II-3: A core can configure itself as an execute core.
- II-4: A core can de-configure itself as an execute core.
- II-5: The `invalidate` signal de-configures the access and execute cores for each FIFO.
- II-6: The `invalidate` signal waits for outstanding memory requests and eventually resets all scratchpad entries to `INV`.

- II-7: Verify which FIFOs have an access core configured at any time using a model.
- II-8: Verify which FIFOs have an execute core configured at any time using a model.
- II-9: Requests entering the CONF pipeline are eventually responded to.
- II-10: Only load requests from a valid FIFO by its configured execute core are recognized as valid load requests.
- II-11: Stage L_0 in the LOAD pipeline is always available for incoming valid load requests.
- II-12: Requests entering the LOAD pipeline are eventually responded to.
- II-13: Only store requests for a valid FIFO by its configured access core are recognized as valid store requests.
- II-14: Stage S_0 in the STORE pipeline is always available for incoming valid store requests.
- II-15: Requests entering the STORE pipeline are eventually responded to.
- II-16: All requests are eventually responded to (summarizes II-9, II-12 and II-15).

14.2.2 Phase II Assumptions

The properties below are assumptions that were needed during Phase II to ensure appropriate inputs to the `dcp_pipe` module. They concern communication between the `dcp_pipe` module and other parts of the system, making them relevant for multiple pipelines.

- II-17: Each core can have up to one outstanding request to the IS tile.
- II-18: L2 will eventually respond to any request.
- II-19: L2 will not respond without a corresponding request.
- II-20: DRAM will eventually respond to any request.
- II-21: DRAM will not respond without a corresponding request.
- II-22: Incoming TLOAD requests specify an 8-byte address to load data from.
- II-23: For each valid load request, there is eventually a corresponding valid store request.
- II-24: For each valid store request, there is eventually a corresponding valid load request.
- II-25: Valid incoming requests on NOC1 will eventually be accepted.
- II-26: Valid incoming requests on NOC2 will eventually be accepted.

15 Phase III: Verification at the dcp Level

15.1 Specification of the dcp Module

The dcp module is a thin wrapper module around an instance of the dcp_pipe module and instances of the required buffers and NOC decoders/encoders, as shown in Figure 14.

On the input side, the dcp module first buffers any messages detected on the 3 NOCs. NOC1 is used for incoming messages from cores, NOC2 is used for incoming L2 data responses, while NOC3 is used for incoming DRAM data responses. Then, one message at a time is decoded appropriately and passed along to the dcp_pipe module, with each field of the message forwarded as an individual signal. NOC3 messages have priority over NOC2 messages, which in turn have priority over NOC1 messages.

On the output side, the dcp module first buffers any outgoing messages provided by the dcp_pipe module. Then, one message at a time is encoded appropriately and passed along to the appropriate NOC, once it is available. NOC1 is used for outgoing L2 data requests, while NOC2 is used for outgoing ACK messages and outgoing DRAM data requests.

15.2 Relevant Properties

This Section presents a high-level description of all the properties verified during Phase III, to ensure that the dcp module abides by the specification presented in Section 15.1. The corresponding SVA code for each of these properties is available in Appendix C. In Section 15.2.2, we also present the assumptions that were needed during Phase III (III-17 and beyond) regarding the correct functionality of modules outside dcp.

15.2.1 Phase III Assertions

The properties below verify that any outgoing messages from DCP are well-formed according to the NOC specifications. Properties III-1 to III-7 concern outgoing messages sent via NOC1, while Properties III-8 to III-15 concern outgoing messages sent via NOC2. Finally, property III-16 verifies the fact that DCP is always available to receive messages via NOC3, i.e. that DCP never blocks the highest-priority NOC.

- III-1: Any message we send via NOC1 remains stable until NOC1 is available.
- III-2: Any message we send via NOC1 has a message ID within range.
- III-3: Any message we send via NOC1 is of a legal type.
- III-4: Any message we send via NOC1 consists of 4 flits.
- III-5: Any message we send via NOC1 correctly specifies its destination core.
- III-6: Any message we send via NOC1 correctly specifies this IS tile as its source.
- III-7: Any message we send via NOC1 requests data of a legal size.
- III-8: Any message we send via NOC2 remains stable until NOC2 is available.
- III-9: Any message we send via NOC2 is of a legal type.
- III-10: Any message we send via NOC2 consists of 1 flit if it ACKs a store request.
- III-11: Any message we send via NOC2 consists of 2 flits if it ACKs a load request.
- III-12: Any message we send via NOC2 consists of 3 flits if it requests data from DRAM.
- III-13: Any ACK message we send via NOC2 correctly specifies its destination core.
- III-14: Any message we send via NOC2 correctly specifies its destination core if it requests data from DRAM.
- III-15: Any message we send via NOC2 correctly specifies this IS tile as its source.
- III-16: DCP never blocks NOC3 by not being ready to receive a message.

15.2.2 Phase III Assumptions

The properties below are assumptions that were needed during Phase III to ensure appropriate inputs to the `dcp` module. Property III-17 assumes that the tile identifier of a DCP instance is stable. The remaining properties assume that any incoming messages are well-formed according to the NOC specifications: properties III-18 to III-23 concern incoming messages received via NOC1; properties III-24 to III-27 concern incoming messages received via NOC2; and properties III-28 to III-31 concern incoming messages received via NOC3. Properties III-32 to III-37 assume that cores, the L2 cache and DRAM conform to their communication protocol with the IS tile. Finally, properties III-38 to III-44 assume the fairness and completeness of incoming messages.

- III-17: The IS tile identifier is stable.
- III-18: Any message we receive via NOC1 remains stable until it has been received.
- III-19: Any message we receive via NOC1 has a message ID within range.
- III-20: Any message we receive via NOC1 is of a legal type.
- III-21: Any message we receive via NOC1 consists of 4 flits.
- III-22: Any message we receive via NOC1 correctly specifies this IS tile as its destination.
- III-23: Any message we receive via NOC1 specifies a valid opcode.
- III-24: Any message we receive via NOC2 has a message ID within range.
- III-25: Any message we receive via NOC2 is of a legal type (L2 data response).
- III-26: Any message we receive via NOC2 consists of 2 flits.
- III-27: Any message we receive via NOC2 correctly specifies this IS tile as its destination.
- III-28: Any message we receive via NOC3 has a message ID within range.
- III-29: Any message we receive via NOC3 is of a legal type (DRAM data response).
- III-30: Any message we receive via NOC3 consists of 2 flits.
- III-31: Any message we receive via NOC3 correctly specifies this IS tile as its destination.
- III-32: L2 will eventually respond to any request.
- III-33: L2 will not respond without a corresponding request.
- III-34: DRAM will eventually respond to any request.
- III-35: DRAM will not respond without a corresponding request.
- III-36: Each core can have up to one outstanding request to the IS tile.
- III-37: No request can arrive from a core that already has an outstanding request to this tile.
- III-38: For each valid store request, there is eventually a corresponding valid load request.
- III-39: For each valid load request, there is eventually a corresponding valid store request.
- III-40: Valid incoming requests on NOC1 will eventually be processed.
- III-41: Valid incoming requests on NOC2 will eventually be processed.
- III-42: If we receive one flit of a message via NOC1, we will eventually also receive the rest.
- III-43: If we receive one flit of a message via NOC2, we will eventually also receive the rest.
- III-44: If we receive one flit of a message via NOC3, we will eventually also receive the rest.

16 Evaluation

Section 16.1 presents the results of this project, alongside some verification quality metrics provided by JasperGold. The rest of this Section presents indicative case studies of bugs discovered through formal verification that led to the reconsideration of DCP design decisions.

16.1 Verification Results

A single-FIFO version of DCP was initially verified, which did not include the concepts of FIFO creation and validity. This is equivalent to considering all scratchpad entries as assigned to FIFO 0, which is always valid, while any other FIFOs are always invalid. All three phases were successfully completed for this design, with a 100 % proof rate across all properties. This provides confidence that, at its core, the current design successfully conforms to its specification.

To verify the multi-FIFO version described in this report, the values $N = 16$ and $F = 2$ were selected, in order to limit the size of the state space that JasperGold had to explore. For this version, Phase I has been successfully completed and Phase II is currently underway, with full proofs for 78% of properties and bounded proofs of between 18 and 32 cycles for the remaining properties after a 24-hour run. As explained in Section 11.2, more complicated designs like the multi-FIFO version take combinatorially longer to formally verify than simpler ones, while the then-current FV Phase also needs to be restarted every time a bug is found and fixed.

JasperGold also provides metrics for the fraction of the total RTL code that was reached and exercised during FV. For Phase II for the version with $N = 16$ and $F = 2$, 100% of the RTL code was reached within 16-24 clock cycles (*reachable*), while 90% of the RTL code was actively checked by one or more properties (*inside the proof core*). Upon manual inspection of the parts of the code that fell outside the proof core, they are exactly the parts that are intentionally not yet targeted by our verification plan because they are not critical and not yet finalized; for example, certain DCP performance counters intended to provide cores with statistics about DCP usage.

Since the hardest-to-reach parts of our code only took 16-24 cycles to reach, and FV exposes the *shortest* trace violating an assertion, bounds of at least 18-32 cycles for our assertions are within a very encouraging range, reducing the probability that JasperGold finds an assertion violation at higher depths and pointing towards a conclusion of Phase II in the near future.

16.2 Case Study 1: Integrating the `dcp_noc1buffer` Module

As shown in Figure 14, the `dcp` module also includes instances of modules from previous work [48], namely buffers and encoders/decoders for each NOC. Even though these modules were thoroughly verified in the context of the system they were originally developed for, Phase III of our FV discovered a bug in the integration of one of them into DCP. In particular, the `dcp_noc1buffer` module did not check whether the buffer it manages was full before accepting new messages.

The `dcp_noc1buffer` module provides a buffer for outgoing messages for NOC1 until NOC1 is available. In the original context for which `dcp_noc1buffer` was developed, the module feeding this buffer had a small cap on the number of simultaneous outstanding messages it could have. Thus, setting the buffer size equal to that cap meant that no check for fullness was necessary.

However, outgoing DCP messages directed at NOC1 are outgoing L2 data requests, as per Section 15.1. In theory, for given patterns of valid store requests arriving to `dcp_pipe`, there could be as many back-to-back simultaneously outstanding L2 data requests as scratchpad entries. Since making the buffer inside `dcp_noc1buffer` as large as the DCP scratchpad would be a wasteful use of resources, a smaller buffer size was chosen instead.

As long as the buffer inside the `dcp_noc1buffer` module remains full because NOC1 is unavailable, the expected behavior would be for the buffer to apply *backpressure* to `dcp_pipe`; a store request responsible for an additional L2 data request should stall in stage S_2 of the STORE pipeline (see Section 14.1.3). However, without the `dcp_noc1buffer` module checking whether the buffer is full, some L2 data requests would simply be overwritten and dropped when the buffer was full. This bug was revealed by JasperGold during Phase II as a violation of property II-12 (and II-16, see Section 14.2), since some load requests could get stuck in the LOAD pipeline indefinitely because the L2 data request for the data they were expecting had been dropped. As such, the `dcp_noc1buffer` module was adjusted to apply backpressure whenever full.

This case study illustrates an important point about reusing modules from past systems in new projects: correctness for a module is always defined with respect to some set of assumptions about its environment. As such, a careful review of the new set of such assumptions is always necessary when reusing a module, since optimizations made in the original context (e.g. not testing for fullness because the buffer size is hand-picked) may no longer be applicable.

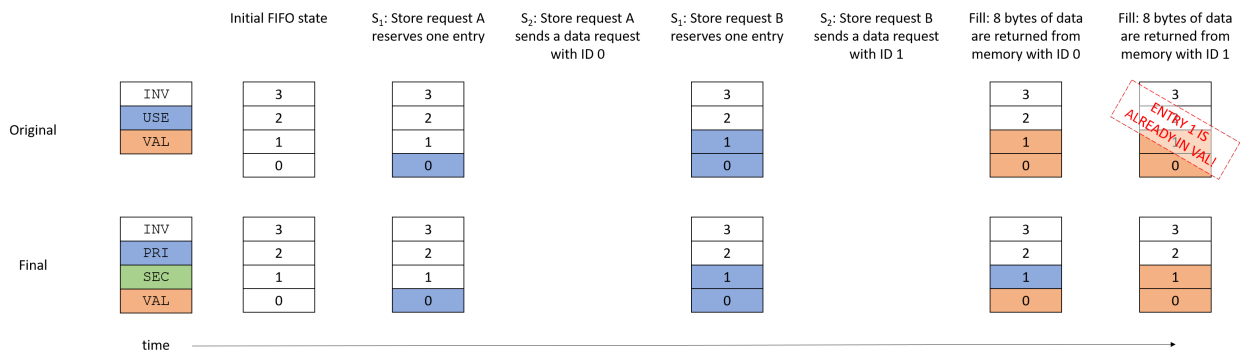

Figure 18: Case study 2: By having a generic USE state instead of differentiating between the PRI and SEC states, we cannot track the expected size of a data response.

16.3 Case Study 2: Distinguishing the PRI and SEC Entry States

The original specification of the dcp_fifo_ctrl module only allowed for 3 distinct states for each scratchpad entry: INV, USE and VAL. While INV and VAL were defined exactly as presented in Section 13.1.1, USE was a combination of PRI and SEC, simply interpreted as "the entry does not currently contain meaningful data, but is involved in an outstanding memory request". However, this created issues since there was no way of tracking how many bytes (4 or 8) were expected as a data response - as mentioned in Section 14.1.4, this information is not encoded in the data response message, which always contains 8 bytes of data, so DCP has to keep track of it on its own side.

A naive approach to determine whether to fill one or two entries with the incoming data was to check whether the "next" entry in the relevant FIFO was also in USE. Certainly, if it was not, then only one entry had been reserved, so only 4 bytes should be kept from the data response. However, as presented in Figure 18, the situation was more complicated when there were more than one outstanding requests; the next entry may actually have been in USE because it was reserved for a different request. If it is filled by erroneously keeping all 8 bytes of the current data response, the next data response would later arrive to find its reserved entry had already been filled.

This scenario was revealed by JasperGold during Phase II as a violation of property I-13 (see Section 13.2), since the later response was trying to fill an entry that was already in VAL . As such, the scratchpad design was modified to distinguish between PRI and SEC states. With this change, by checking that the "next" entry in the relevant FIFO is in SEC, DCP can ensure that that entry was reserved as part of the same request.

16.4 Case Study 3: Waiting for Data Responses After Using `invalidate`

As explained in Section 13.1.2, if an IS tile receives the `invalidate` signal, the access and the execute cores of all FIFOs are immediately de-configured, but the tile waits for any outstanding requests to memory before resetting all scratchpad entries to `INV` and all FIFOs to invalid. No FIFOs can be created on this tile until this invalidation process is complete. At first glance, the design choice to wait may seem counter-intuitive: why would the party issuing the `invalidate` signal have to wait for memory responses before the tile is reset and available for use once again?

Assume instead that all effects of the `invalidate` signal took place immediately, which was the original design decision. Assume further that the tile had at least one outstanding request to memory when the `invalidate` signal was received. As constrained by properties III-32 and III-34 (see Section 15.2.2), both the L2 cache and DRAM are guaranteed by their specifications to respond to outstanding requests, so there are two possible cases.

First, it might be that the data response is the first message received by the tile after the `invalidate` signal. In this case, property I-13 (see Section 13.2) will be violated, since the response is trying to fill an entry that is in `INV` and the data will be dropped. This scenario would not be catastrophic on its own, since the dropped data was presumably no longer needed if the `invalidate` signal was used. However, problems arise when the data response arrives so late that the tile has already been re-configured post invalidation, like in the following sequence of events:

1. FIFO 0 is created and cores `ac0` and `ex0` are configured as its access and execute cores.
2. Core `ac0` reserves entry 0 of FIFO 0 and a request is sent to memory for 4 bytes from $\texttt{addr}_0$.
3. The tile receives the `invalidate` signal, all effects of which take place immediately.
4. FIFO 0 is created again and cores `ac1` and `ex1` are configured as its access and execute cores.
5. Core `ac1` reserves entry 0 of FIFO 0 and a request is sent to memory for 4 bytes from $\texttt{addr}_1$.
6. Data originating from $\texttt{addr}_0$ arrives from memory, intended for entry 0 of FIFO 0. The entry is currently in the `PRI` state, so it is filled with the data and changed to the `VAL` state.
7. The events below happen in either order:

    i. Core `ex1` reads entry 0 of FIFO 0, expecting the 4 bytes from address $\texttt{addr}_1$, but instead gets the 4 bytes from address $\texttt{addr}_0$. The entry is changed to the `INV` state.

ii. Data originating from $\texttt{addr}_1$ arrives from memory, intended for entry 0 of FIFO 0, but the entry is not currently in the PRI or SEC state.

Note that this sequence of events leads to core ex1 receiving incorrect data, in addition to property I-13 being violated (see Section 13.2); the later response was trying to fill an entry that was either in VAL or in INV, depending on the relative order of events 7i and 7ii. The only way to avoid it would be for the tile to somehow know at step 6 that it should only accept data from $\texttt{addr}_1$ for this entry at this time. However, the address from which the data originated is not included in the data response message. Even if it were, each FIFO entry in PRI or SEC would have needed to keep track of the memory address from which it is expecting data, which would be very expensive in terms of required state elements.

As such, in order to avoid such cases, the design was modified so that tiles will instead wait for any outstanding requests to memory before being available again for new FIFO creation.

17 Conclusions and Future Work

As evident from this work, formal verification is a powerful approach in our efforts to build reliable and correct systems. Where simulation-based verification requires large suites of test cases to be written, which is both resource-intensive and provides limited completeness guarantees, formal verification mathematically proves system correctness across the entire space of possible system behaviors based only on a simple set of properties.

Moreover, using FV can help uncover bugs within orders of magnitude fewer cycles compared to simulation. Where simulation might incidentally expose a problematic scenario as part of a longer trace, FV tools are actively searching for the *shortest* trace that causes an assertion violation. As such, it is also much easier to pinpoint and fix a bug by examining this shorter trace.

For the DCP subsystem specifically, we performed formal verification in three phases, consistent with the nested nature of the new modules being tested. As evident by the metrics provided by JasperGold, presented in Section 16.1, the set of properties that we verified thoroughly checks our design. We have successfully verified a single-FIFO version of our design across all three phases, which gives us confidence in the basic structure of the design, while we have also made significant progress towards fully verifying the multi-FIFO version described in this report.

Although we focused on the DCP subsystem for this project, it is only a small part of the overall DECADES chip. A natural next step would be to extend our formal verification to other parts of the chip. For example, the IS tile currently has no support for virtual memory, even though it eventually will. As such, once the relevant components have been designed (e.g. a TLB), we can write properties to formally verify them. Beyond the IS tile, we can also use FV for other individual parts of the DECADES chip, like caches or particular pipelines.

With so many additional opportunities to use FV, another future direction we could take is that of automating parts of the FV process that we followed. For example, we could automate the generation of SVA transaction models, like the one used for properties II-18 and II-19 (see Appendix B). Such models are generally similar in structure, so we could use a script to generate them based on an XML specification of the protocol format, further increasing the efficiency of our FV workflow.

Still, since DECADES is a large design, some simulation is ultimately inevitable; as can be deduced from Section 11.2, performing FV *at the chip level* would be infeasible, due to the combinatorial state-space explosion that would ensue. Even so, there are 3 important benefits to using FV until that point. First, our unit-level testing will be smoother, since the shorter violation traces exposed by FV make bugs easier to find and fix. Second, we will get formal guarantees about the internal workings of each module, letting us focus our chip-level simulation on interfaces. Third, we will be able to reuse some properties written during FV, especially assumptions made by each module about other modules, as assertions during chip-level simulation, reducing the additional work required at that stage.

A SVA Code Concerning the `dcp_fifo_ctrl` Module

As noted in Section 13.2, properties I-13 and beyond partially depend on the correct functionality of modules outside `dcp_fifo_ctrl`. As such, these properties were phrased as assumptions in Phase I and changed to assertions for Phases II and III.

```
//////// Declarations

// A symbolic scratchpad entry index and the ‘‘previous’’ entry in the same FIFO
// (with wraparound).
wire [FC_GLOBL_IDX-1:0] symb_idx = 'x;
wire [FC_GLOBL_IDX-1:0] symb_prev = (symb_idx == fifo_base[fc_decr_idx]) ?
   fifo_top_r[fc_decr_idx] - 1'd1 : symb_idx - 1'd1;
am__symb_idx_stable: assume property($stable(symb_idx));

// A symbolic FIFO index.
wire [FC_FIFOS_IDX-1:0] symb_fifo = 'x;
am__symb_fifo_stable: assume property($stable(symb_fifo));

//////// Phase I Assertions

//// Concerning Scratchpad Entries

//// Transitions from the INV state

// I-1: An entry that is not currently in use cannot be valid in the next clock
// cycle (it must first get reserved and then get filled).
as__st_legal_out_of_inv: assert property (!data_use[symb_idx] |=>
   !data_val[symb_idx]);

//// Transitions from the PRI/SEC states

// I-2: An entry in the PRI or SEC state cannot directly transition to the INV
```

```
// state (it must first be filled and then be read), unless we reset the system.
as__st_legal_out_of_use: assert property (data_use[symb_idx] &&
    !data_val[symb_idx] |=> data_use[symb_idx]);

//// Transitions from the VAL state

// I-3: A valid entry is also in use (bookkeeping correctness check).
as__st_val_implies_use: assert property (data_val[symb_idx] |->
    data_use[symb_idx]);

// I-4: A valid entry can never be in PRI or SEC in the next clock cycle.
as__st_legal_out_of_val: assert property (data_val[symb_idx] |=>
    (!data_use[symb_idx] || data_val[symb_idx]));

// I-5: A valid entry stays valid if we don't read it, reset the system or use
// invalidate.
as__still_valid_not_consumed1: assert property (data_val[symb_idx] &&
    !fc_decr_len && !(fc_decr_entry == symb_idx) && !invalidate |=>
    data_val[symb_idx]); // 4-byte read
as__still_valid_not_consumed2: assert property (data_val[symb_idx] &&
    fc_decr_len && !(fc_decr_entry == symb_idx) && !(fc_decr_entry == symb_prev)
    && !invalidate |=> data_val[symb_idx]); // 8-byte read

// I-6: A valid entry is no longer valid once we read it.
as__not_valid_if_consumed: assert property (data_val[symb_idx] && (fc_decr_entry
    == symb_idx) && fc_decr_val && fc_decr_res |=> !data_val[symb_idx]);

//// Concerning FIFOs

// I-7: We only confirm entry reservation requests for non-full FIFOs.
as__fifo_full_no_write: assert property (fifo_full |-> !fc_incr_res);
```

```
// I-8: If we reject an entry reservation request for a valid FIFO, either
// the FIFO was full, or 2 entries where requested while only 1 was not in use.
as__no_res_then_fifo_full: assert property (fc_fifo_val[fc_incr_idx] &&
    !fc_incr_res |-> fifo_full || fc_incr_len);

// I-9: We only successfully load data from non-empty FIFOs.
as__fifo_empty_no_read: assert property (fifo_empty |-> !fc_decr_res);

// I-10: When storing data, the tail is incremented by the correct amount.
as__fifo_incr_len: assert property ((fc_incr_hsk && (fc_incr_idx == symb_fifo)
    && !c0_invalidate) |=> (((fifo_tail_r[symb_fifo] -
    $past(fifo_tail_r[symb_fifo])) % fifo_size) == 1 + $past(fc_incr_len)));

// I-11: When loading data, the head is incremented by the correct amount.
as__fifo_decr_len: assert property ((fc_decr_hsk && (fc_decr_idx == symb_fifo)
    && !c0_invalidate) |=> (((fifo_head_r[symb_fifo] -
    $past(fifo_head_r[symb_fifo])) % fifo_size) == 1 + $past(fc_decr_len)));

//// Concerning Data Integrity

//// Model for I-12.

// Set to 1 if the scratchpad entry at symb_idx has been filled.
logic stored;
// If stored is 1, contains the data with which the entry at symb_idx was filled.
logic [FC_DATA_SIZE-1:0] data_model;

// Is this entry being filed by a 4-byte fill, or with the 4 least significant
// bytes of an 8-byte fill?
wire fill0 = fc_fill_val && fc_fill_entry == symb_idx;

// Is this entry being filed with the 4 most significant bytes of an 8-byte fill?
```

```
wire fill1 = fc_fill_val && (((fc_fill_entry + 1)%FC_GLOBL_SIZE) == symb_idx) &&
    (data_st_r[symb_idx] == 2'b10);

// Model state update logic.
always_ff @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
      stored <= 1'b0;
      data_model <= 1'b0;
    end else begin
      if( fill0 || fill1 ) begin
        stored <= 1'b1;
        data_model <= fill0 ? fc_fill_data[FC_DATA_SIZE-1:0] :
            fc_fill_data[FC_DATA_SIZE*2-1:FC_DATA_SIZE];
      end
    end
end

////

// I-12: Any data loaded from a scratchpad entry is the same data that was
// stored earlier.
as__data_integrity_check: assert property(stored && (fc_decr_entry == symb_idx)
    && fc_decr_hsk |-> (fc_decr_data0 == data_model));

//////// Phase I Assumptions

// I-13: Any valid incoming fill data is only intended for entries
// in use, but not valid.
as__fill_iface : assert property (fc_fill_val |-> data_use[fc_fill_entry] &&
    !data_val[fc_fill_entry]);

// I-14: Any valid FIFO creation requests concern FIFOs that were not already
```

```
// valid and have ensured that enough scratchpad entries are available.
wire fc_add_size_correct = (fc_add_size % 4 == 0) && (fc_add_size > 0) &&
    (fc_add_size <= 2**FC_ENTRY_IDX);
as__add_fifo : assume property (fc_add_val |-> !fifo_val_r[fc_add_idx] &&
    fc_add_size_correct);

// I-15: We only receive entry reservation requests for valid FIFOs.
as__incr_fifo : assert property (fc_incr_val |-> fifo_val_r[fc_incr_idx]);

// I-16: We only receive load requests for valid FIFOs.
as__decr_fifo : assert property (fc_decr_val |-> fifo_val_r[fc_decr_idx]);
```

B SVA Code Concerning the `dcp_pipe` Module

As noted in Section 14.2, properties II-17 and beyond partially depend on the correct functionality of modules outside `dcp_pipe`. As such, these properties were phrased as assumptions in Phase II and changed to assertions for Phase III. MSHRID refers to the message ID.

```
//////// Declarations

// A symbolic message ID (MSHRID).
wire [`DCP_MSHRID_WIDTH-1:0] symb_mshrid = 'x;
am__symb_mshrid : assume property ($stable(symb_mshrid));

// A symbolic core ID.
wire [`PACKET_HOME_ID_WIDTH-1:0] symb_core = 'x;
am__symb_core : assume property ($stable(symb_core));

// A symbolic FIFO index.
wire [`FC_FIFO_IDX-1:0] symb_fifo = 'x;
am__symb_fifo_stable: assume property($stable(symb_fifo));
```

```
// The relevant FIFO index: the lowest-index formerly-invalid FIFO for FIFO
// creation, or the FIFO specified by the incoming message otherwise.
wire fifo_idx = fc_add_hsk ? fc_add_idx : s0_fifo;

//////// Phase II Assertions

//// CONF Pipeline

// II-1: A core can configure itself as an access core.
as__aconf_works: assert property (aconfiguring && (noc1decoder_dcp_src ==
    symb_core) |=> (conf_access_id_r[$past(fifo_idx)] == symb_core));

// II-2: A core can de-configure itself as an access core.
as__adeconf_works: assert property (adeconfiguring |=>
    !is_aconfigured_r[$past(fifo_idx)]);

// II-3: A core can configure itself as an execute core.
as__econf_works: assert property (econfiguring && (noc1decoder_dcp_src ==
    symb_core) |=> (conf_execute_id_r[$past(fifo_idx)] == symb_core));

// II-4: A core can de-configure itself as an execute core.
as__edeconf_works: assert property (edeconfiguring |=>
    !is_econfigured_r[$past(fifo_idx)]);

// II-5: The invalidate signal de-configures the access and execute cores for
// each FIFO.
as__invalidate_works_cores: assert property (invalidate |=>
    !is_aconfigured_r[symb_fifo] && !is_econfigured_r[symb_fifo]);

// II-6: The invalidate signal waits for outstanding memory requests and
// eventually resets all scratchpad entries to INV.
```

```
as__invalidate_works_fifo: assert property (invalidate && !(count_fifo_use ==
    '0) |-> s_eventually((count_fifo_use == '0)));

//// Model for II-7 and II-8.

// Keep track of FIFOs with an access core configured.
reg [`FC_FIFO_SIZE-1:0] is_aconfigured_r;

// Keep track of FIFOs with an execute core configured.
reg [`FC_FIFO_SIZE-1:0] is_econfigured_r;

// Model state update logic.
always_ff @(posedge clk) begin
    if (!rst_n || invalidate) begin
        is_aconfigured_r <= {`FC_FIFO_SIZE{1'b0}};
        is_econfigured_r <= {`FC_FIFO_SIZE{1'b0}};
    end
    else begin
        if (adeconfiguring) begin
            is_aconfigured_r[fifo_idx] <= 1'b0;
        end
        if (edeconfiguring) begin
            is_econfigured_r[fifo_idx] <= 1'b0;
        end
        if (aconfiguring && !invalidate) begin
            is_aconfigured_r[fifo_idx] <= 1'b1;
        end
        if (econfiguring && !invalidate) begin
            is_econfigured_r[fifo_idx] <= 1'b1;
        end
    end
end
```

```
////

// II-7: Verify which FIFOs have an access core configured at any time using a
// model.
as__aconf_model_match: assert property(is_aconfigured_r[symb_fifo] ==
    c0_aconfigured_r[symb_fifo]);

// II-8: Verify which FIFOs have an execute core configured at any time using a
// model.
as__econf_model_match: assert property(is_econfigured_r[symb_fifo] ==
    c0_econfigured_r[symb_fifo]);

// II-9: Requests entering the CONF pipeline are eventually responded to.
as__conf_eventual_resp: assert property(noc1decoder_dcp_hsk &&
    (noc1decoder_dcp_mshrid == symb_mshrid) && dcp_pipe.noc1decoder_dcp_conf |->
    s_eventually(c1_val && (dcp_pipe.c1_mshr == symb_mshrid)));

//// LOAD Pipeline

// II-10: Only load requests from a valid FIFO by its configured execute core
// are recognized as valid load requests.
as__load_is_econfed: assert property (((noc1decoder_dcp_reqtype ==
    `DCP_REQ_LOAD) && is_econfigured_r[fifo_idx] && dcp_pipe.cons_op &&
    noc1decoder_dcp_src == conf_execute_id_r[fifo_idx]) ==
    dcp_pipe.noc1decoder_dcp_load);

// II-11: Stage L_0 in the LOAD pipeline is always available for incoming valid
// load requests.
as__load_ready: assert property (noc1decoder_dcp_val &&
    dcp_pipe.noc1decoder_dcp_load |-> dcp_pipe.l0_rdy);
```

```
// II-12: Requests entering the LOAD pipeline are eventually responded to.
as__load_eventual_resp: assert property(noc1decoder_dcp_hsk &&
    (noc1decoder_dcp_mshrid == symb_mshrid) && dcp_pipe.noc1decoder_dcp_load |->
    s_eventually(dcp_pipe.l2_val && (dcp_pipe.l2_mshr == symb_mshrid)));

//// STORE Pipeline

// II-13: Only store requests for a valid FIFO by its configured access core
// are recognized as valid store requests.
as__store_is_aconfed: assert property (((noc1decoder_dcp_reqtype ==
    `DCP_REQ_STORE) && is_aconfigured_r[fifo_idx] && dcp_pipe.store_op_legal &&
    noc1decoder_dcp_src == conf_access_id_r[fifo_idx]) ==
    dcp_pipe.noc1decoder_dcp_store);

// II-14: Stage S_0 in the STORE pipeline is always available for incoming valid
// store requests.
as__store_ready: assert property (noc1decoder_dcp_val &&
    dcp_pipe.noc1decoder_dcp_store |-> dcp_pipe.s0_rdy);

// II-15: Requests entering the STORE pipeline are eventually responded to.
as__store_eventual_resp: assert property(noc1decoder_dcp_hsk &&
    (noc1decoder_dcp_mshrid == symb_mshrid) && dcp_pipe.noc1decoder_dcp_store |->
    s_eventually(dcp_pipe.s3_val && (dcp_pipe.s3_mshr == symb_mshrid)));

//// Any individual pipeline

// II-16: All requests are eventually responded to (combines II-9, II-12 and
// II-15).
as__eventual_resp_end2end: assert property(noc1decoder_dcp_hsk &&
    (noc1decoder_dcp_mshrid == symb_mshrid) |-> s_eventually(dcp_noc2buffer_val
    && (dcp_noc2buffer_mshrid == symb_mshrid)));
```

```
//////// Phase II Assumptions

//// Model for II-17.

// Keep track of the outstanding requests to the IS tile per core.
logic [NUM_CORES-1:0][1:0] core_outstanding_req_r;

// Are we receiving a new request from some core in the same cycle that we are
// responding to an existing request from that core?
wire core_idx_equal = noc1decoder_dcp_src == dcp_noc2buffer_homeid;

// Model state update logic.
always_ff @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    core_outstanding_req_r <= {NUM_CORES{2'b0}};
  end else begin
    if (noc1decoder_dcp_hsk && !(dcp_noc2buffer_ack_hsk && core_idx_equal)) begin
      core_outstanding_req_r[noc1decoder_dcp_src] <=
          core_outstanding_req_r[noc1decoder_dcp_src] + 1'b1;
    end
    if (dcp_noc2buffer_ack_hsk && !(noc1decoder_dcp_hsk && core_idx_equal)) begin
      core_outstanding_req_r[dcp_noc2buffer_homeid] <=
          core_outstanding_req_r[dcp_noc2buffer_homeid] - 1'b1;
    end
  end
end

////

// II-17: Each core can have up to one outstanding request to the IS tile.
as__core_max_1_outstanding: assert property(core_outstanding_req_r[symb_core] <
    2'b10);
```

```
//// Model for II-18 and II-19.

// Keep track of the outstanding requests to L2.
logic [MSHRIDS - 1:0] l2_outstanding_req_r;

// Model state update logic.
always_ff @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    l2_outstanding_req_r <= {MSHRIDS{1'b0}};
  end else begin
    if (dcp_noc1buffer_hsk) begin
      l2_outstanding_req_r[dcp_noc1buffer_mshrid] <= 1'b1;
    end
    if (noc2decoder_dcp_hsk) begin
      l2_outstanding_req_r[noc2decoder_dcp_mshrid] <= 1'b0;
    end
  end
end

////

// II-18: L2 will eventually respond to any request.
as__get_l2_noc_response_eventually : assert
    property(l2_outstanding_req_r[symb_mshrid] |->
    s_eventually(noc2decoder_dcp_hsk && (noc2decoder_dcp_mshrid == symb_mshrid)));

// II-19: L2 will not respond without a corresponding request.
as__not_get_l2_response_if_not_req : assert
    property(l2_outstanding_req_r[symb_mshrid] == '0 |-> !(noc2decoder_dcp_hsk &&
    (noc2decoder_dcp_mshrid == symb_mshrid)));
```

```
//// Model for II-20 and II-21.

// Keep track of the outstanding requests to DRAM.
logic [MSHRIDS - 1:0] dram_outstanding_req_r;

// Model state update logic.
always_ff @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    dram_outstanding_req_r <= {MSHRIDS{1'b0}};
  end else begin
    if (dcp_noc2buffer_tload_hsk) begin
      dram_outstanding_req_r[dcp_noc2buffer_mshrid] <= 1'b1;
    end
    if (noc3decoder_dcp_hsk) begin
      dram_outstanding_req_r[noc3decoder_dcp_mshrid] <= 1'b0;
    end
  end
end

////

// II-20: DRAM will eventually respond to any request.
as__get_dram_noc_response_eventually : assert
    property(dram_outstanding_req_r[symb_mshrid] |->
    s_eventually(noc3decoder_dcp_hsk && (noc3decoder_dcp_mshrid == symb_mshrid)));

// II-21: DRAM will not respond without a corresponding request.
as__not_get_dram_response_if_not_req : assert
    property(dram_outstanding_req_r[symb_mshrid] == '0 |-> !(noc3decoder_dcp_hsk
    && (noc3decoder_dcp_mshrid == symb_mshrid)));
```

```
// II-22: Incoming TLOAD requests specify an 8-byte address to load data from.
wire is_tload = (noc1_opcode == `DCP_TLOAD32_OP || noc1_opcode ==
    `DCP_TLOAD64_OP);
as__tloads_are_64: assert property(is_tload |-> noc1decoder_dcp_size ==
    `MSG_DATA_SIZE_8B);

// II-23: For each valid load request, there is eventually a corresponding
// valid store request.
wire valid_store = noc1decoder_dcp_val && dcp_pipe.noc1decoder_dcp_store;
as__noc1_store_fairness : assert property(fc_decr_val |->
    s_eventually(valid_store));

// II-24: For each valid store request, there is eventually a corresponding
// valid load request.
wire valid_load = noc1decoder_dcp_val && dcp_pipe.noc1decoder_dcp_load;
as__noc1_load_fairness : assert property(fc_incr_val |->
    s_eventually(valid_load));

// II-25: Valid incoming requests on NOC1 will eventually be accepted.
as__noc1_ready_fairness : assert property(dcp_noc1buffer_val |->
    s_eventually(dcp_noc1buffer_rdy));

// II-26: Valid incoming requests on NOC2 will eventually be accepted.
as__noc2_ready_fairness : assert property(dcp_noc2buffer_val |->
    s_eventually(dcp_noc2buffer_rdy));
```

C SVA Code Concerning the `dcp` Module

As noted in Section 15.2, properties III-17 and beyond partially depend on the correct functionality of modules outside `dcp`. As such, these properties were phrased as assumptions in Phase III. MSHRID refers to the message ID.

```
//////// Models

//// Model for outgoing messages.

// Keep track of which flit of an outgoing message we are currently sending.
reg [1:0] noc1_out_state;
reg [1:0] noc2_out_state;

// Name states accordingly.
wire noc1_out_head1 = dcp_dst_vr_noc1_val && noc1_out_state == 2'b00;
wire noc1_out_head2 = dcp_dst_vr_noc1_val && noc1_out_state == 2'b01;
wire noc1_out_head3 = dcp_dst_vr_noc1_val && noc1_out_state == 2'b10;
wire noc1_out_data1 = dcp_dst_vr_noc1_val && noc1_out_state == 2'b11;
wire noc2_out_head1 = dcp_dst_vr_noc2_val && noc2_out_state == 2'b00;
wire noc2_out_head2 = dcp_dst_vr_noc2_val && noc2_out_state == 2'b01;
wire noc2_out_head3 = dcp_dst_vr_noc2_val && noc2_out_state == 2'b10;
wire noc2_out_data1 = dcp_dst_vr_noc2_val && noc2_out_state == 2'b11;

// Keep track of the length of the message currently being sent via NOC2
// (number of additional flits).
reg [1:0] noc2_out_state_len;
wire [1:0] noc2_len = dcp_dst_vr_noc2_dat[`MSG_LENGTH];
wire [1:0] noc2_out_state_nxt = (noc2_out_state + 1'b1) % (noc2_out_head1 ?
    noc2_len + 1'b1 : noc2_out_state_len + 1'b1);

// Model state update logic.
always @ (posedge clk) begin
    if (!rst_n) begin
        noc1_out_state <= 2'b0;
        noc2_out_state <= 2'b0;
        noc2_out_state_len <= 2'b0;
```

```
    end else begin
        if (dcp_dst_vr_noc1_hsk) noc1_out_state <= noc1_out_state + 1'b1;
        if (dcp_dst_vr_noc2_hsk) noc2_out_state <= noc2_out_state_nxt;
        if (dcp_dst_vr_noc2_hsk && noc2_out_head1) noc2_out_state_len <= noc2_len;
    end
end

////

//// Model for incoming messages.

// Keep track of which flit of an incoming message we are currently receiving.
reg [1:0] noc1_in_state;
reg       noc2_in_state;
reg       noc3_in_state;

// Name states accordingly.
wire noc1_in_head1 = noc1_in_state == 2'b00;
wire noc1_in_head2 = noc1_in_state == 2'b01;
wire noc1_in_head3 = noc1_in_state == 2'b10;
wire noc1_in_data1 = noc1_in_state == 2'b11;
wire noc2_in_head = noc2_in_state == 1'b0;
wire noc2_in_data = noc2_in_state == 1'b1;
wire noc3_in_head = noc3_in_state == 1'b0;
wire noc3_in_data = noc3_in_state == 1'b1;

// Model state update logic.
always @ (posedge clk) begin
    if (!rst_n) begin
        noc2_in_state <= 1'b0;
        noc3_in_state <= 1'b0;
        noc1_in_state <= 2'b0;
```

```
    end else begin
        if (src_dcp_vr_noc2_hsk) noc2_in_state <= noc2_in_state + 1'b1;
        if (src_dcp_vr_noc3_hsk) noc3_in_state <= noc3_in_state + 1'b1;
        if (src_dcp_vr_noc1_hsk) noc1_in_state <= noc1_in_state + 1'b1;
    end
end

////

//// Model for outstanding requests to L2.

// Keep track of the outstanding requests to L2.
logic [MSHRIDS - 1:0] l2_outstanding_req_r;

// Model state update logic.
always_ff @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    l2_outstanding_req_r <= {MSHRIDS{1'b0}};
  end else begin
    if (dcp_dst_vr_noc1_hsk && noc1_out_head1) begin
      l2_outstanding_req_r[noc1_out_mshrid] <= 1'b1;
    end
    if (src_dcp_vr_noc2_hsk && noc2_in_head) begin
      l2_outstanding_req_r[noc2_in_mshrid] <= 1'b0;
    end
  end
end

////

//// Model for outstanding requests to DRAM.
```

```
// Keep track of the outstanding requests to DRAM.
logic [MSHRIDS - 1:0] dram_outstanding_req_r;

// Model state update logic.
always_ff @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    dram_outstanding_req_r <= {MSHRIDS{1'b0}};
  end else begin
    if (dcp_dst_vr_noc2_tload_hsk) begin
      dram_outstanding_req_r[noc2_out_mshrid] <= 1'b1;
    end
    if (src_dcp_vr_noc3_hsk && noc3_in_head) begin
      dram_outstanding_req_r[noc3_in_mshrid] <= 1'b0;
    end
  end
end

////

//// Model for outstanding requests to the IS tile per core.

// Keep track of the outstanding requests to the IS tile per core.
logic [NUM_CORES-1:0][1:0] core_outstanding_req_r;

// Are we receiving a new request from some core in the same cycle that we are
// responding to an existing request from that core?
wire core_idx_equal = src_dcp_vr_noc1_coreid == dcp_dst_vr_noc2_coreid;

// Model state update logic.
always_ff @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    core_outstanding_req_r <= {NUM_CORES{2'b0}};
```

```
  end else begin
    if (src_dcp_vr_noc1_head3_hsk && !(dcp_dst_vr_noc2_ack_hsk &&
        core_idx_equal)) begin
      core_outstanding_req_r[src_dcp_vr_noc1_coreid] <=
          core_outstanding_req_r[src_dcp_vr_noc1_coreid] + 1'b1;
    end
    if (dcp_dst_vr_noc2_ack_hsk && !(src_dcp_vr_noc1_head3_hsk &&
        core_idx_equal)) begin
      core_outstanding_req_r[dcp_dst_vr_noc2_coreid] <=
          core_outstanding_req_r[dcp_dst_vr_noc2_coreid] - 1'b1;
    end
  end
end

////

//// Model for remembering request type while receiving subsequent flits.

// Is the incoming message a store per the NOC definition? (All IS tile requests
// are formatted as stores per the NOC definition).
wire noc1_in_store = src_dcp_vr_noc1_dat[`MSG_TYPE] == `DCP_REQ_STORE;
reg noc1_in_was_store;

// Is the incoming message a store or load per the DCP definition?
wire noc1_in_load_op_legal = (noc1_opcode == `DCP_CONS_OP);
wire noc1_in_store_op_legal = (noc1_opcode == `DCP_AMO_OP_ADD) || (noc1_opcode
    == `DCP_AMO_OP_AND) || (noc1_opcode == `DCP_AMO_OP_OR) || (noc1_opcode ==
    `DCP_AMO_OP_XOR) || (noc1_opcode == `DCP_AMO_OP_MAX) || (noc1_opcode ==
    `DCP_AMO_OP_MAXU) || (noc1_opcode == `DCP_AMO_OP_MIN) || (noc1_opcode ==
    `DCP_AMO_OP_MINU) || (noc1_opcode == `DCP_AMO_OP_SWAP) || (noc1_opcode ==
    `DCP_AMO_OP_CAS1) || (noc1_opcode == `DCP_AMO_OP_CAS2) || (noc1_opcode ==
    `DCP_PROD_OP) || (noc1_opcode == `DCP_TLOAD32_OP) || (noc1_opcode ==
```

```
    `DCP_TLOAD64_OP);
reg noc1_in_was_st_op_legal;
reg noc1_in_was_ld_op_legal;

// Model state update logic.
always @ (posedge clk) begin
    if (!rst_n) begin
        noc1_in_was_store <= 1'b0;
        noc1_in_was_st_op_legal <= 1'b0;
        noc1_in_was_ld_op_legal <= 1'b0;
    end else begin
        if (src_dcp_vr_noc1_head1_hsk) noc1_in_was_store <= noc1_in_store;
        if (src_dcp_vr_noc1_head2_hsk) begin
            noc1_in_was_st_op_legal <= noc1_in_store_op_legal;
            noc1_in_was_ld_op_legal <= noc1_in_load_op_legal;
        end
    end
end

////

//////// Phase III Assertions

//// NOC1

// III-1: Any message we send via NOC1 remains stable until NOC1 is available.
as__stability_noc1_data_out: assert property (dcp_dst_vr_noc1_val &&
    !dcp_dst_vr_noc1_rdy |=> $stable(dcp_dst_vr_noc1_dat));

// III-2: Any message we send via NOC1 has a message ID within range.
as__noc1_out_mshr: assert property (noc1_out_head1 |->
    dcp_dst_vr_noc1_dat[`MSG_MSHRID] < 2**`DCP_MSHRID_WIDTH);
```

```
// III-3: Any message we send via NOC1 is of a legal type.
wire noc1_out_type_correct = (dcp_dst_vr_noc1_dat[`MSG_TYPE] ==
    `MSG_TYPE_CAS_REQ) || (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_SWAP_REQ)
    || (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_AMO_ADD_REQ) ||
    (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_AMO_AND_REQ) ||
    (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_AMO_OR_REQ) ||
    (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_AMO_XOR_REQ) ||
    (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_AMO_MAX_REQ) ||
    (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_AMO_MAXU_REQ) ||
    (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_AMO_MIN_REQ) ||
    (dcp_dst_vr_noc1_dat[`MSG_TYPE] == `MSG_TYPE_AMO_MINU_REQ);
as__noc1_out_type: assert property (noc1_out_head1 |-> noc1_out_type_correct);

// III-4: Any message we send via NOC1 consists of 4 flits.
as__noc1_out_msg_len: assert property (noc1_out_head1 |->
    dcp_dst_vr_noc1_dat[`MSG_LENGTH] == 3);

// III-5: Any message we send via NOC1 correctly specifies its destination core.
as__noc1_out_coreid: assert property (noc1_out_head1 |-> dcp_dst_vr_noc1_coreid
    == noc1_out_dst_coreid);

// III-6: Any message we send via NOC1 correctly specifies this IS tile as its
// source.
as__noc1_out_coreid_src : assert property (noc1_out_head3 |->
    dcp_dst_vr_noc1_coreid == tileid);

// III-7: Any message we send via NOC1 requests data of a legal size.
wire data_size_correct = dcp_dst_vr_noc1_dat[`MSG_DATA_SIZE_] ==
    `MSG_DATA_SIZE_4B || dcp_dst_vr_noc1_dat[`MSG_DATA_SIZE_] ==
    `MSG_DATA_SIZE_8B;
as__noc1_out_size: assert property (noc1_out_head2 |-> data_size_correct);
```

```
//// NOC2

// III-8: Any message we send via NOC2 remains stable until NOC2 is available.
as__stability_noc2_data_out: assert property (dcp_dst_vr_noc2_val &&
    !dcp_dst_vr_noc2_rdy |=> $stable(dcp_dst_vr_noc2_dat));

// III-9: Any message we send via NOC2 is of a legal type.
wire noc2_out_type_correct = (dcp_dst_vr_noc2_dat[`MSG_TYPE] ==
    `DCP_REQ_STORE_ACK) || (dcp_dst_vr_noc2_dat[`MSG_TYPE] == `DCP_REQ_LOAD_ACK)
    || (dcp_dst_vr_noc2_dat[`MSG_TYPE] == `MSG_TYPE_LOAD_MEM);
as__noc2_out_type: assert property (noc2_out_head1 |-> noc2_out_type_correct);

// III-10: Any message we send via NOC2 consists of 1 flit if it ACKs a store
// request.
as__noc2_out_st_len: assert property (noc2_out_head1 && noc2_out_st_ack |->
    dcp_dst_vr_noc2_dat[`MSG_LENGTH] == 0);

// III-11: Any message we send via NOC2 consists of 2 flits if it ACKs a load
// request.
as__noc2_out_ld_len: assert property (noc2_out_head1 && noc2_out_ld_ack |->
    dcp_dst_vr_noc2_dat[`MSG_LENGTH] == 1);

// III-12: Any message we send via NOC2 consists of 3 flits if it requests data
// from DRAM.
as__noc2_out_tl_len: assert property (noc2_out_head1 && noc2_out_tl_req |->
    dcp_dst_vr_noc2_dat[`MSG_LENGTH] == 2);

// III-13: Any ACK message we send via NOC2 correctly specifies its destination
// core.
wire noc2_out_dst_correct = |core_outstanding_req_r[dcp_dst_vr_noc2_coreid];
as__noc2_out_coreid: assert property (noc2_out_head1 && (noc2_out_st_ack ||
```

```
    noc2_out_ld_ack) |-> noc2_out_dst_correct);

// III-14: Any message we send via NOC2 correctly specifies its destination core
// if it requests data from DRAM.
as__noc2_out_tl_coreid: assert property (noc2_out_head1 && noc2_out_tl_req |->
    dcp_dst_vr_noc2_coreid == {1'b1,{`PACKET_HOME_ID_WIDTH-1{1'b0}}} );

// III-15: Any message we send via NOC2 correctly specifies this IS tile as its
// source.
as__noc2_out_coreid_src: assert property (noc2_out_head3 |->
    dcp_dst_vr_noc2_coreid == tileid);

//// NOC3

// III-16: DCP never blocks NOC3 by not being ready to receive a message.
as__noc3_resp_never_block: assert property (dcp.noc3_data_val |->
    dcp.noc3_data_ack);

//////// Phase III Assumptions

// III-17: The IS tile identifier is stable.
am__tile_id_stable : assume property ($stable(tileid));

//// NOC1

// III-18: Any message we receive via NOC1 remains stable until it has been
    received.
am__stability_noc1_data_in: assume property (src_dcp_vr_noc1_val &&
    !src_dcp_vr_noc1_rdy |=> $stable(src_dcp_vr_noc1_dat));

// III-19: Any message we receive via NOC1 has a message ID within range.
am__noc1_in_mshr: assume property (noc1_in_head1 |->
```

```
    src_dcp_vr_noc1_dat[`MSG_MSHRID] < 2**`DCP_MSHRID_WIDTH);

// III-20: Any message we receive via NOC1 is of a legal type.
am__noc1_in_type: assume property (noc1_in_head1 |->
    (src_dcp_vr_noc1_dat[`MSG_TYPE] == `DCP_REQ_STORE) ||
    (src_dcp_vr_noc1_dat[`MSG_TYPE] == `DCP_REQ_LOAD));

// III-21: Any message we receive via NOC1 consists of 4 flits.
am__noc1_in_msg_len: assume property (noc1_in_head1 |->
    src_dcp_vr_noc1_dat[`MSG_LENGTH] == 3);

// III-22: Any message we receive via NOC1 correctly specifies this IS tile as
// its destination.
am__noc1_in_coreid: assume property (noc1_in_head1 |-> src_dcp_vr_noc1_coreid ==
    tileid);

// III-23: Any message we receive via NOC1 specifies a valid opcode.
am__noc1_in_addr: assume property(noc1_in_head2 |-> (noc1_opcode !=
    `DCP_AMO_OP_CAS1) && (noc1_opcode != `DCP_AMO_OP_CAS2));

//// NOC2

// III-24: Any message we receive via NOC2 has a message ID within range.
am__noc2_in_mshr: assume property(noc2_in_head |->
    src_dcp_vr_noc2_dat[`MSG_MSHRID] < 2**`DCP_MSHRID_WIDTH);

// III-25: Any message we receive via NOC2 is of a legal type (L2 data response).
am__noc2_in_type: assume property (src_dcp_vr_noc2_dat[`MSG_TYPE] ==
    `MSG_TYPE_DATA_ACK);

// III-26: Any message we receive via NOC2 consists of 2 flits.
am__noc2_in_msg_len: assume property(noc2_in_head |->
```

```
    src_dcp_vr_noc2_dat[`MSG_LENGTH] == 1);

// III-27: Any message we receive via NOC2 correctly specifies this IS tile as
// its destination.
am__noc2_in_msg_coreid: assume property(noc2_in_head |-> src_dcp_vr_noc2_coreid
    == tileid);

//// NOC3

// III-28: Any message we receive via NOC3 has a message ID within range.
am__noc3_in_mshr: assume property(noc3_in_head |->
    src_dcp_vr_noc3_dat[`MSG_MSHRID] < 2**`DCP_MSHRID_WIDTH);

// III-29: Any message we receive via NOC3 is of a legal type (DRAM data
// response).
am__noc3_in_type: assume property (src_dcp_vr_noc3_dat[`MSG_TYPE] ==
    `MSG_TYPE_LOAD_MEM_ACK);

// III-30: Any message we receive via NOC3 consists of 2 flits.
am__noc3_in_msg_len: assume property(noc3_in_head |->
    src_dcp_vr_noc3_dat[`MSG_LENGTH] == 1);

// III-31: Any message we receive via NOC3 correctly specifies this IS tile as
// its destination.
am__noc3_in_msg_coreid : assume property(src_dcp_vr_noc3_coreid == tileid);

//// Responsiveness of memory and cores.

generate for (genvar i = 0; i < MSHRIDS; i = i + 1) begin : l2_noc_assumes
    // III-32: L2 will eventually respond to any request.
    am__get_l2_noc_response_eventually : assume property(l2_outstanding_req_r[i]
        |-> s_eventually(src_dcp_vr_noc2_hsk && (src_dcp_vr_noc2_dat[`MSG_MSHRID]
```

```
        == i)));

    // III-33: L2 will not respond without a corresponding request.
    am__not_get_l2_response_if_not_req : assume property(l2_outstanding_req_r[i]
        == '0|-> !(src_dcp_vr_noc2_hsk && (src_dcp_vr_noc2_dat[`MSG_MSHRID] ==
        i)));

    // III-34: DRAM will eventually respond to any request.
    am__get_dram_noc_response_eventually : assume
        property(dram_outstanding_req_r[i] |-> s_eventually(src_dcp_vr_noc3_hsk
        && (src_dcp_vr_noc3_dat[`MSG_MSHRID] == i)));

    // III-35: DRAM will not respond without a corresponding request.
    am__not_get_dram_response_if_not_req : assume
        property(dram_outstanding_req_r[i] == '0 |-> !(src_dcp_vr_noc3_hsk &&
        (src_dcp_vr_noc3_dat[`MSG_MSHRID] == i)));
  end
endgenerate

generate for (genvar i = 0; i < NUM_CORES; i = i + 1) begin : core_noc_assumes
    // III-36: Each core can have up to one outstanding request to the IS tile.
    am__core_max_1_outstanding1: assume property(core_outstanding_req_r[i] <
        2'b10);

    // III-37: No request can arrive from a core that already has an
    // outstanding request to this tile.
    am__core_max_1_outstanding2: assume property(core_outstanding_req_r[i] >
        2'b00 |-> !(src_dcp_vr_noc1_val && noc1_in_head3 &&
        (src_dcp_vr_noc1_coreid == i)));
  end
endgenerate
```

```
//// Fairness and completeness.

// III-38: For each valid store request, there is eventually a corresponding
// valid load request.
wire valid_load = src_dcp_vr_noc1_val && noc1_in_head3 &&
    noc1_in_was_ld_op_legal && !noc1_in_was_store && (src_dcp_vr_noc1_coreid ==
    dcp_pipe.conf_execute_id_r) && dcp_pipe.c0_econfigured_r;
am__noc1_load_fairness : assume property(dcp_pipe.fc_incr_val |->
    s_eventually(valid_load));

// III-39: For each valid load request, there is eventually a corresponding
// valid store request.
wire valid_store = src_dcp_vr_noc1_val && noc1_in_head3 &&
    noc1_in_was_st_op_legal && noc1_in_was_store && (src_dcp_vr_noc1_coreid ==
    dcp_pipe.conf_access_id_r) && dcp_pipe.c0_aconfigured_r;
am__noc1_store_fairness : assume property(dcp_pipe.fc_decr_val |->
    s_eventually(valid_store));

// III-40: Valid incoming requests on NOC1 will eventually be processed.
am__noc1_ready_fairness : assume property(dcp_dst_vr_noc1_val |->
    s_eventually(dcp_dst_vr_noc1_rdy));

// III-41: Valid incoming requests on NOC2 will eventually be processed.
am__noc2_ready_fairness : assume property(dcp_dst_vr_noc2_val |->
    s_eventually(dcp_dst_vr_noc2_rdy));

// III-42: If we receive one flit of a message via NOC1, we will eventually
// also receive the rest.
am__noc1_in_completeness: assume property(!noc1_in_head1 |->
    s_eventually(noc1_in_head1));

// III-43: If we receive one flit of a message via NOC2, we will eventually
```

```
// also receive the rest.
am__noc2_in_completeness: assume property(!noc2_in_head |->
    s_eventually(noc2_in_head));

// III-44: If we receive one flit of a message via NOC3, we will eventually
// also receive the rest.
am__noc3_in_completeness: assume property(!noc3_in_head |->
    s_eventually(noc3_in_head));
```

References

[1] T. Melissaris, M. Markakis, K. Shaw, and M. Martonosi, "PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing," *submitted to the 53rd IEEE/ACM International Symposium on Microarchitecture*, 2020.

[2] *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.

[3] *The GeForce 16 Series Graphics Cards are Here*, 2019. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/gtx-1660-ti/.

[4] M. D. Hill and V. J. Reddi, *Accelerator-level Parallelism*, 2019. arXiv: 1907.02064 [cs.DC].

[5] 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, Santa Barbara, CA, USA: ACM, 2016, pp. 100–113, ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908114. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908114.

[6] 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, Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 635–646, ISBN: 978-1-4799-6998-2. DOI: 10.1109/MICRO.2014.38. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.38.

[7] 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, Portland, Oregon: ACM, 2015, pp. 388–400, ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750378. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750378.

[8] Inria, *The diy software suite*, Jul. 2019. [Online]. Available: http://diy.inria.fr/.

[9] 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. DOI: 10.1007/978-3-642-19835-9_5.

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

[11] 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. DOI: 10.1109/tc.1979.1675439.

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

[13] 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. DOI: 10.1007/s10766-011-0176-3. [Online]. Available: https://doi.org/10.1007/s10766-011-0176-3.

[14] 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. DOI: 10.1007/978-3-642-03359-9_27.

[15] 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. DOI: 10.1145/1785414.1785443. [Online]. Available: https://doi.org/10.1145/1785414.1785443.

[16] 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," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 577–591, 2015. DOI: 10.1145/2775054.2694391. [Online]. Available: http://www.cs.princeton.edu/~ts20/files/asplos2015.pdf.

[17] J. Alglave, "A formal hierarchy of weak memory models," *Formal Methods in System Design*, vol. 41, no. 2, pp. 178–210, Jun. 2012. DOI: 10.1007/s10703-012-0161-5.

[18] 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, 7:1–7:74, Jul. 2014, ISSN: 0164-0925. DOI: 10.1145/2627752. [Online]. Available: http://doi.acm.org/10.1145/2627752.

[19] 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, Seattle, Washington, USA: ACM, 1990, pp. 2–14, ISBN: 0-89791-366-3. DOI: 10.1145/325164.325100. [Online]. Available: http://doi.acm.org/10.1145/325164.325100.

[20] 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, Seattle, Washington, USA: ACM, 1990, pp. 15–26, ISBN: 0-89791-366-3. DOI: 10.1145/325164.325102. [Online]. Available: http://doi.acm.org/10.1145/325164.325102.

[21] *SPARC architecture manual, version 9*, 1994.

[22] *Alpha architecture reference manual*, 1992.

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

[24] 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, Edinburgh, UK: Springer-Verlag, 2010, pp. 258–272. DOI: 10.1007/978-3-642-14295-6_25. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_25.

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

[26] 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, St. Petersburg, FL, USA: ACM, 2016, pp. 649–662, ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837643. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837643.

[27] 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, Paris, France: ACM, 2017, pp. 175–189, ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009850. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009850.

[28] 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, Mumbai, India: ACM, 2015, pp. 209–220, ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676995. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676995.

[29] 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.

[30] 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, ISSN: 1045-9219. DOI: 10.1109/TPDS.2003.1199067. [Online]. Available: https://doi.org/10.1109/TPDS.2003.1199067.

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

[32] 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.*, Jun. 2004, pp. 114–123. DOI: 10.1109/ISCA.2004.1310768.

[33] 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)*, Mar. 2016, pp. 618–630. DOI: 10.1109/HPCA.2016.7446099.

[34] 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, Waikiki, Hawaii: ACM, 2015, pp. 26–37, ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830782. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830782.

[35] 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, ISSN: 0362-1340. DOI: 10.1145/3093336.3037719. [Online]. Available: http://doi.acm.org/10.1145/3093336.3037719.

[36] 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, Cambridge, Massachusetts: ACM, 2017, pp. 463–476, ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3124536. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124536.

[37] 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, Fukuoka, Japan: IEEE Press, 2018, pp. 788–801, ISBN: 978-1-5386-6240-3. DOI: 10.1109/MICRO.2018.00069. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00069.

[38] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically Comparing Memory Consistency Models," in *ACM SIGPLAN Notices*, ACM, vol. 52, 2017, pp. 190–204.

[39] 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, Xi'an, China: ACM, 2017, pp. 661–675, ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037723. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037723.

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

[41] 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, ISSN: 0362-1340. DOI: 10.1145/2775054.2694391. [Online]. Available: http://doi.acm.org/10.1145/2775054.2694391.

[42] A. Adir and G. Shurek, "Generating Concurrent Test-programs with Collisions for Multiprocessor 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–, ISBN: 0-7803-7655-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1114283.1114492.

[43] T. Rondeau, *Software Defined Hardware (SDH)*. [Online]. Available: https://www.darpa.mil/program/software-defined-hardware.

[44] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, "OpenDCP: Plug-n-play Memory Latency Tolerance for Simple Off-the-shelf Cores," *submitted to the 53rd IEEE/ACM International Symposium on Microarchitecture*, 2020.

[45] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures," in *The International Symposium on Microarchitecture (MICRO)*, ACM, 2015, pp. 191–203.

[46] ——, "Decoupling Data Supply from Computation for Latency-Tolerant Communication in Heterogeneous Architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, Jun. 2017, ISSN: 1544-3566. DOI: 10.1145/3075620. [Online]. Available: http://doi.acm.org/10.1145/3075620.

[47] J. E. Smith, "Decoupled access/execute computer architectures," in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society Press, vol. 10, 1982, pp. 112–119.

[48] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "OpenPiton: An Open Source Manycore Research Framework," in *ASPLOS*, ACM, 2016, pp. 217–232.

[49] *IEEE Std 1800.2-2017: IEEE Standard for Universal Verification Methodology Language Reference Manual*. IEEE, 2017.

[50] E. Seligman, T. Schubert, and M. V. Achutha Kiran Kumar, *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann, 2015.

[51] *What is Scrum?* [Online]. Available: `https://www.scrum.org/resources/what-is-scrum`.

[52] *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012): IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE, 2018.

[53] *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005): IEEE Standard for Property Specification Language (PSL)*. IEEE, 2010.

[54] *JasperGold Formal Property Verification App*. [Online]. Available: `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html`.