Check for updates

PipeGen: Automated Transformation of a Single-Core Pipeline into a Multicore Pipeline for a Given Memory Consistency Model

An Qi Zhang
University of Utah
United States of America
an.qi.zhang@utah.edu

Andrés Goens
University of Amsterdam
Netherlands
a.goens@uva.nl

Nicolai Oswald
Nvidia
United States of America
mail@nicolai-oswald.de

Tobias Grosser
University of Cambridge
United Kingdom
tobias.grosser@cst.cam.ac.uk

Daniel Sorin
Duke University
United States of America
sorin@ee.duke.edu

Vijay Nagarajan
University of Utah
United States of America
vijay@cs.utah.edu

Abstract

Designing a pipeline for a multicore processor is difficult. One major challenge is designing it such that the pipeline correctly enforces the intended memory consistency model (MCM). We have developed the PipeGen design automation tool to allow architects to start with a single core pipeline that only enforces single-threaded correctness and automatically transform it to enforce a given MCM. Our key innovation is a set of compiler-like transformations that codify three different ways of enforcing memory ordering at the pipeline. We have validated that PipeGen correctly enforces the ARMv8 and x86TSO MCMs on three distinct pipeline implementations, using litmus tests with the Murphi model checker.

CCS Concepts

• **Computer systems organization** → **Multicore architectures**.

Keywords

Memory Consistency Model, Computer Architecture, Microarchitecture, Programming Language, Compiler

ACM Reference Format:

An Qi Zhang, Andrés Goens, Nicolai Oswald, Tobias Grosser, Daniel Sorin, and Vijay Nagarajan. 2024. PipeGen: Automated Transformation of a Single-Core Pipeline into a Multicore Pipeline for a Given Memory Consistency Model. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3656019.3676889

1 Introduction

Designing a modern, high-performance processor pipeline is a difficult challenge. In the pursuit of performance, cores often seek additional improvements by executing instructions out-of-order. Out-of-order execution must not affect single-threaded functionality, though, and thus microarchitects use structures like the reorder buffer (ROB) and load-store queue (LSQ) to ensure the illusion of in-order behavior for a given hardware thread.

In addition to single-threaded correctness, another key challenge is ensuring that a processor comprised of multiple high-performance cores maintains the desired memory consistency model (MCM). It is very difficult to reason about the possible interleavings of reads (loads) and writes (stores) across threads on different cores and whether they are allowed by the MCM. An architect may design an optimization but fail to realize its full potential on the enforced MCM [1, 20], or more worryingly, make a design error that leads to the desired MCM no longer being enforced [10, 16, 17].

In this work, we create the PipeGen design automation tool to address this design challenge. Specifically, PipeGen enables an architect to design a pipeline in an MCM-oblivious fashion, needing only to enforce single-threaded correctness. The architect specifies the single-core design using a new microarchitectural domain specific language (DSL) that we have developed for this purpose. PipeGen then transforms that pipeline into a multicore pipeline that enforces the specified MCM, as illustrated in Figure 1. A key insight is that we can achieve this automation as *code analysis and transformations* in a DSL, borrowing methods from compiler design. PipeGen currently uses transformations that encode three different ways of enforcing memory ordering in the pipeline.

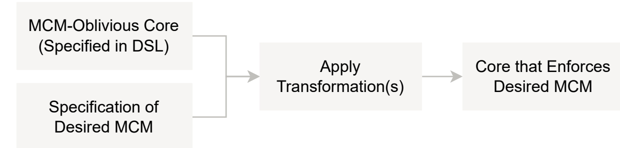

Figure 1: Automation goal: Given a pipeline in our DSL that correctly enforces single-threaded correctness, and a desired MCM, PipeGen automatically creates a pipeline that enforces the desired MCM.

PipeGen enables microarchitects to focus on designing their core, while offloading the burden of correctly implementing the MCM to our automation tool. PipeGen additionally decouples the implementation of the ISA and MCM, reducing the work necessary to enforce a different MCM. In the history of microarchitectures and MCMs, it has not been uncommon to move between ISAs that use

This work is licensed under a Creative Commons Attribution International 4.0 License.

PACT '24, October 14–16, 2024, Long Beach, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676889

different MCMs, as Apple has done between IBM, x86TSO, and now ARMv8. PipeGen creates the possibility of designing an efficient microarchitecture, and being able to reuse it across different MCMs by using our algorithms to enforce the desired MCM.

Our DSL is tailored for specifying the structures that comprise a microarchitecture. The primary structures are queues that contain one or more entries, such as the reorder buffer (ROB) and load-store queue (LSQ). As such, we refer to our DSL as "A Queue Language" (AQL). Within these queues, the architect specifies the controller state machines that describe the behaviors of these structures. Crucially, we require the architect to identify and label certain key memory instruction states to facilitate PipeGen's automation.

PipeGen's output is a transformed core, described in AQL, that supports the specified MCM. Our key contribution is a set of three compiler-like transformations implementing three different methods of enforcing memory ordering (in-order memory instructions, load replay, and invalidation tracking). While the methods are well known, our innovation here is to codify this domain knowledge into transformations, allowing us to instantiate these on any single-core pipeline expressible in AQL. With the variety of different pipeline microarchitectures being implemented across the different types of processors, this style of design automation has the potential for broad applicability and impact.

We verify that PipeGen correctly produces these cores by automatically translating the output AQL into the language of the Murphi model checker [9] and performing model checking on specific so-called litmus tests [2]. We have verified PipeGen for a variety of core models and MCMs. While other backends are possible – e.g., to hardware description languages such as Verilog – we leave them for future work.

While there has been much work on verifying that a given pipeline core correctly enforces the specified MCM, the most recent of which is the *-Check suite of tools [14, 17], our point of departure is that our work is a correct-by-construction approach to generating pipelines. This correct-by-construction methodology has seen a recent resurgence for generating coherence protocols [24] and single-core processors [25] but ours is the first to automate the generation of MCM implementations to our knowledge. We discuss related work more extensively in Section 7.

Our contributions are:

- We have created PipeGen, a methodology and a tool for automatically transforming a single-core pipeline into a multicore pipeline that adheres to a given MCM.
- PipeGen consists of a DSL called AQL and a set of compiler-like analyses and transformations. Through these transformations, we codify, for the first time, three well-known methods for enforcing memory orderings. This codification allows us to instantiate these methods automatically on any given single-core pipeline expressible in AQL.
- We have validated PipeGen on 3 different pipeline implementations to enforce 2 different memory models (x86TSO and ARMv8) using 3 different methods for enforcing memory ordering (in-order issue, load replay or invalidation-tracking). These were validated in the Murphi model checker exhaustively running litmus tests. Our results show that PipeGen produces multicore pipelines that correctly support the specified MCMs.

2 Background: System Model and MCMs

In this section, we explain our single-core pipeline model and provide the necessary background on MCMs and how they can be enforced at the pipeline level.

2.1 Single-Core Pipeline Model

We consider a fairly generic pipeline model that supports out-of-order execution of instructions while providing precise exceptions through in-order (i.e., in program order) fetch and commit. The pipeline consists of multiple structures that manage the out-of-order execution. Different cores use different structures, but commonly used examples include the reorder buffer (ROB), load-store queue (LSQ), physical register file, and post-commit store buffer (SB).

Because MCMs concern the ordering of memory operations (loads, stores, fences), PipeGen's transformations focus on the structures that manage the out-of-order execution of memory operations. Even in a single-core pipeline, management of memory operations can be complicated, due to the desire to speculatively execute loads before all previous (in program order) memory operations have completed. To ensure that PipeGen is broadly applicable, we consider pipelines with three quite different designs for managing memory operations as case studies. There are, of course, many designs because of the many possible trade-offs between structure size, complexity, performance, and power, and we considered these three designs to showcase the versatility of PipeGen. In each of these designs there is an issue queue (IQ) into which all of the instructions are inserted in program order. The designs differ from each other based on what happens to a memory instruction upon dispatch from the issue queue.

Design 1: LQ/SQ/WB. Design-1 uses three distinct structures to manage memory instructions: store queue (SQ), post-commit write buffer (WB), and load queue (LQ). At dispatch from the issue queue, a store is allocated an address-tagged entry in the SQ. When a store executes, it writes the value into its SQ entry. When a store commits, it frees its SQ entry and writes to the tail of the WB. Similarly, at dispatch, a load is allocated an address-tagged entry in the LQ. At execute, a load searches the SQ for the most recent store to the same address that is older than the load. If no match is found in the SQ, the load accesses the WB for the most recent store to the same address; if none is found, the load accesses the memory system (i.e., starting with the L1 data cache). To detect misspeculation in which a load speculatively reads an address before an older store (which is possible because an older store's address might not yet have been available when the load was ready to execute), when a store executes it checks the LQ for younger loads that have already executed; a match indicates misspeculation.

This design provides the most performance of the three we consider. However, all three structures require associative searches. A load must be able to find address-matching entries in the SQ and WB, and a store must be able to find address-matching entries in the LQ.

Design 2: LSQ. Design-2 uses a single load-store queue (LSQ) and no WB. The LSQ serves the purpose of both the LQ and SQ

Table 1: x86TSO memory orderings. Row labels are earlier in program order than column labels. AA indicates Any Address, and SA Same Address.

	Load	Store	mfence
Load	AA	AA	AA
Store	SA	AA	AA
mfence	AA	AA	AA

in Design-1. Because this LSQ must be associatively searched (by both loads and stores), it cannot be made larger than either the LQ or SQ from Design-1, and thus can manage fewer in-flight memory operations. But Design-2 is less expensive that Design-1. This design also does not have a WB. Stores issue to the memory system when they reach the top of the ROB, and so stores are performed in order.

Design 3: LB. Design-3 allows loads to speculatively enter a load buffer (LB) as soon as their addresses are available. (We use a different term here – buffer rather than a queue – to reinforce that the unit is simply a staging area for executing the loads.) Because addresses might be resolved in an arbitrary order, loads in this design can potentially execute out-of-order, and loads may not receive data from older stores.

We assume single-core pipelines have the necessary instructions for providing software-directed ordering (e.g., mfence) but they are implemented as NOPs until PipeGen transforms the pipeline.

2.2 MCMs

A multicore pipeline must preserve both single-threaded correctness *and* enforce the architecture's MCM, where an MCM defines the legal apparent orderings of memory operations across all threads. Many consistency models exist, from strongly-ordered models like sequential consistency (SC) and x86TSO to weak models like release consistency (RC) and ARMv8. We briefly discuss two widely used models we focus on in this paper: TSO and ARMv8.

2.2.1 x86TSO. The x86TSO MCM is similar to SC, but relaxes Store → Load ordering across different addresses. This relaxation permits the use of the WB in Design-1, which would violate the stricter SC. When software wants Store → Load ordering to be enforced, it must insert an mfence instruction between the store and the load. The mfence enforces ordering between memory instructions by ensuring older instructions have finished executing before younger instructions execute. With an mfence between a store and load, the load must stall to execute in-order with the store or speculatively execute and then check if it misspeculated. x86TSO orderings are shown above in Table 1.

2.2.2 ARMv8. The ARMv8 MCM relaxes all orderings, only keeping same address dependencies as they are required for single-threaded correctness. When ordering is required, it can be added with load acquires, store releases, and fences. Fences, called Data Memory Barriers (DMB), come in several varieties, including DMB SY (orders all loads and stores), DMB LD (orders Load → Load and Load → Store), and DMB ST (orders Store → Store). Load acquires and store releases are annotated versions of loads and stores, respectively, that also enforce some orderings: specifically the load acquire orders all of the memory operations following it in program order, whereas the store release orders all of the memory operations before it. ARMv8 orderings are shown in Table 2.

Table 2: ARMv8 memory orderings. Row labels are earlier in program order than column labels. AA = Any Address, SA = Same Address, and No indicates an ordering is not enforced.

	LD	LDA	ST	STR	DMB SY	DMB ST	DMB LD
LD	SA	SA	SA	AA	AA	No	AA
LDA	AA	AA	AA	AA	AA	No	AA
ST	SA	SA	SA	AA	AA	AA	No
STR	AA	AA	AA	AA	AA	AA	No
DMB SY	AA	AA	AA	AA	AA	AA	AA
DMB ST	No	No	AA	AA	AA	AA	AA
DMB LD	AA	AA	AA	AA	AA	AA	AA

2.3 MCM Enforcement

Architects have developed mechanisms for *manually* transforming an out-of-order single-core pipeline such that it does not violate the desired MCM. We consider three MCM enforcement mechanisms in this work: In-Order Memory Instructions, Load-Replay [6], and Invalidation Tracking [11]. These are the mechanisms that PipeGen will *automatically* implement when performing its transformations.

2.3.1 In-Order Memory Instructions. One mechanism to order memory instructions is to execute them in order. To order memory instruction type X before memory instruction type Y, an instruction of type Y must stall execution until all older instructions of type X have completed. For example, our Design-1 has a post-commit WB that enables loads to be reordered with respect to older stores. To prevent that reordering, we could require a load to stall until the WB is empty (i.e., all older stores have completed).

2.3.2 Load-Replay. With load-replay [6] a load may speculatively execute out-of-order, and this speculation is checked by replaying the load at commit time. If the replayed load's value matches the initial load value, it has correctly speculated. If not, the new value is written and older instructions are squashed, as they may have used the misspeculated value.

2.3.3 Invalidation Tracking. Invalidation tracking [11] is an alternative mechanism to avoid the need to replay loads at commit. A core observes the incoming cache coherence traffic—specifically, invalidation messages—to identify which addresses have been written by other cores. Any loads that have speculatively read from those addresses may have misspeculated. They, as well as older instructions, are then squashed. Similar to the load-replay mechanism, invalidation handling also enforces Store → Load if there is no WB, since the load must re-execute to again read an updated value.

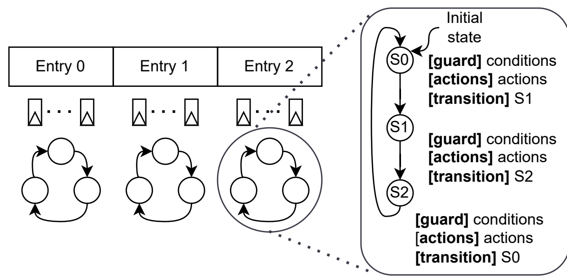

Figure 2: Illustration of a queue structure with state fields shown as registers (boxes with triangles).

3 Input to PipeGen

The input to PipeGen is an MCM-oblivious single-core pipeline. In theory, an architect could specify the core pipeline using any language for expressing finite state machines. HDLs like Verilog or BlueSpec [21] or even state-machine languages like Murphi [9], for example, would all suffice. However, none of these general-purpose languages would make it easy for an automated tool like PipeGen to perform its compiler-like analysis and transformations. We could either mandate a restricted, stylized version of one of these languages or use a domain-specific language (DSL), and we have chosen the latter.

3.1 Abstract Model of Pipeline

We have developed a DSL that enables relatively high-level specifications that focus on functionality more than cycle-accurate behavior. The DSL allows the architect to model the core pipeline as an interconnected group of queue-based structures that each have one or more entries. Because our DSL describes queue-based structures, we call it A Queue Language (AQL). Structures can communicate with each other by sending messages or signals.

Commonly used pipeline structures include the physical register file, reorder buffer (ROB), instruction queue (IQ), load queue (LQ), store queue (SQ), load-store-queue (LSQ), post-commit write buffer (WB), etc. Structures with multiples entries may have an entry ordering, such as FIFO.

Each structure entry is logically a state machine. Each entry has a state (that is initialized to a specified initial state) that consists of one or more fields. An event (i.e., the arrival of a message from another structure) can cause an action and possibly a transition of that entry to a new state. An event satisfies one or more conditions that guard actions/transitions, as illustrated in Figure 2. We illustrate the state machine for Design-1's LQ in Figure 3 which is self-explanatory.

3.2 Memory Instructions: Instruction States and Sub-Operations

To faciliate PipeGen's analysis and transformation algorithms, AQL considers every memory instruction to start in an initial *instruction state* and undergo a sequence of *sub-operations* that change its state.

3.2.1 Instruction States. AQL employs a canonical set of instruction states, each of which is a keyword in AQL, and a memory operation is in one of these states at all times. The initial state

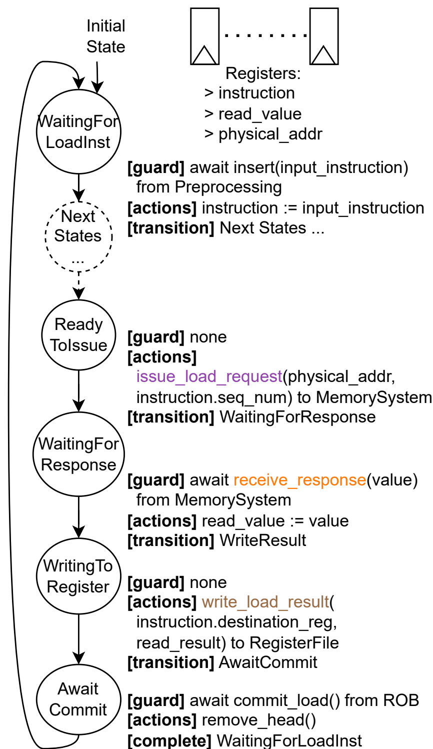

Figure 3: The state machine and state fields of Design-1's LQ entries. Each LQ entry first awaits a load to be inserted in state WaitingForLoadInst. After receiving a load, the entry progresses to performing sub-operations (highlighted in color) such as issuing the load, waiting for it's response, and writing it's result before waiting to be committed. Some states are simplified away, represented by a dotted line.

of a load or store is **ReadyToDispatch**. The sequence of subsequent states for a load that is not misspeculated are: **Dispatched, ReadyToIssue, WaitingForResponse, WritingToRegister**, and **ReadyToCommit**. If the load is misspeculated in either WaitingForResponse or ReadyToCommit, its state changes to an earlier state in the sequence depending on the core design. The ordered sequence of subsequent states for a store are: **Dispatched, StoreValueReady, ReadyToCommit, ReadytoIssue** and **WaitingForResponse**. The ordered sequence for a fence is: **ReadyToDispatch, Dispatched** and **ReadyToCommit**.

An instruction state transition can either be a progression (i.e., proceeding in sequence order) or a reset (i.e., returning to an earlier state in the sequence). This ordering enables PipeGen to find memory operations that are older (i.e., have not progressed as far in their instruction state sequences). Architects can define additional, non-canonical states for their own purposes, but any user-defined state between two canonical states is effectively equal to the previous canonical state.

3.2.2 Sub-operations. AQL considers every memory operation to perform a subset of a canonical set of sub-operations, each of which is a keyword in AQL. These sub-operations are: **dispatch,**

issue_load_request, receive_response, issue_write_request, write_load_result, and **commit**.

Explicitly using these canonical instruction states and sub-operations is critical to PipeGen. At its heart, enforcing memory ordering in the pipeline is all about controlling when and how these sub-operations occur. As we discuss later, adding stalls or replays (or other logic to enforce an MCM) requires PipeGen to identify when to issue to the memory system, when to replay loads, etc.

3.3 AQL Programming Requirements

PipeGen requires the architect to specify instruction states and sub-operations such that it can perform its analyses and transformations. There are several programming guidelines that must be followed. The programmer must:

- Label the canonical sub-operations. PipeGen then identifies the canonical instruction states corresponding to the sub-operations, thereby labeling the canonical instruction states. One of the primary motivations for developing AQL instead of using an existing hardware description language (HDL) is that existing HDLs would not necessarily make instruction states explicit. (They could be implicitly derived from the state of the pipeline, but that would greatly complicate PipeGen.)
- Label a transition with the appropriate label, described in Section 3.2.1, and illustrated in Figure 3.
- Handle instruction misspeculation.
- Label the message that the single-core design uses for squashing misspeculation.

3.4 Case Study: AQL for the LQ in Design 1

In Listing 1, we provide a snippet of AQL code to provide a sense of what it looks like. The snippet includes the code that describes two canonical instruction states in Design-1's LQ. In each state, the user specifies what event (i.e., message or signal) the structure is waiting for before it transitions to its next instruction state.

```
state ReadyToIssue {
  listen { // MemSys: MemorySystem
    MemSys.issue_load_request(physical_addr,
                              instruction.seq_num);
    transition WaitingForResponse;
  } handle squash() from ROB {
    reset WaitingForLoadInst;
  }
}
state WaitingForResponse {
  listen {
    await {
      when receive_response(val) from MemSys {
        read_value = val;
        transition WritingToRegister;
      }
    }
  } handle squash() from ROB {
    reset WaitingForLoadInst;
  }
}
```

Listing 1: Example AQL code describing Design-1's LQ states *ReadyToIssue* and *WaitingForResponse*. ReadyToIssue performs the sub-operation *issue_load_request* and transitions to WaitingForReponse. WaitingForResponse waits for the memory system to respond with the load's value in the sub-operation *receive_response*, then transitions to WriteToRegister. Both states shown are wrapped in a listen-handle block, to handle misspeculation by resetting the entry's state to WaitingForLoadInst.

4 Transformations

Akin to a compiler, the core of PipeGen consists of algorithms for analyzing the input design and transforming it to enforce the desired MCM. In this section, we describe how PipeGen automatically adds the three MCM enforcement mechanisms presented in Section 4.

The choice of which transformations to perform depends on both the user's desired consistency model and the user's preference for the type of MCM enforcement mechanism.

4.1 In-Order Memory Instructions

Consider two memory instructions, M1 and M2, where M1 is before M2 in program order. To enforce in-order execution of two memory instructions M1 → M2, the pipeline must stall M2 until M1 has completed. This enforcement of in-order execution can be used to provide whatever orderings are required by the desired MCM.

To implement this transformation, PipeGen must:

- Identify the M2 instruction state in which M2 should stall.
- Identify the M1 instruction state in which M2 should un-stall.
- Add logic to stall and then un-stall M2 in these identified states.

4.1.1 Identifying M2 State in Which M2 Should Stall. Every type of memory instruction has a canonical instruction state in which it is about to access the memory system, and we refer to this instruction state as the *memory access state*. For a load or store, the access state is ReadyToIssue. For a Fence, the access state is ReadyToCommit. The access state of M2 is the state in which M2 should stall to avoid being ordered ahead of M1. Intuitively, if M2 stalls before it interacts with the memory system, it cannot appear in memory order before M1. Because AQL requires labeling of canonical states in the input pipeline model, it is easy for PipeGen to identify the memory access state for M2.

But blindly choosing the access state as the state in which M2 should stall could lead to deadlock, if we are not careful. Consider a pipeline in which instructions of type M2 reside in a dedicated structure while waiting to access the memory system. Deadlock can arise if this structure fills with younger instructions and an older instruction cannot enter it. To avoid this situation, PipeGen checks if instructions of type M2 enter this M2-specific structure in program order. If not, PipeGen searches backwards through the sequence of structures through which instructions of type M2 traverse, and it has M2 stall at the latest structure in this sequence for which instructions of type M2 are inserted in program order.

4.1.2 Identifying M1 State in Which M2 Should Un-Stall. M2 cannot un-stall until M1 has completed its memory operation and thus PipeGen needs to identify M1's *post-memory completion states*. The

post-memory completion states refer to the set of states after the memory instruction has globally performed. For a load and a store, the post-completion states are all states after WaitingForResponse—i.e., all states after a memory response has been received. For a fence, the post-completion states are all states after ReadyToCommit. Once again, because these instruction states are labeled in AQL, it is simple for PipeGen to find their successors. Specifically, PipeGen extracts the *instruction state graph* from the input AQL code. As discussed in Section 3.2.1, in AQL each instruction has a set of states that it can be in and transitions between these states—we call this an instruction state graph. PipeGen then performs a reachability analysis to compute M1's post-completion states.

4.1.3 Adding Logic to Stall M2 in its Memory Access State and Unstall it in M1's Memory Completion State. When M2 is in its memory access state, the pipeline must stall M2 if M1 is in any instruction state prior to its memory completion state. Thus, PipeGen makes the following transformations:

- Adds logic to the structure that holds M2 when M2 is in its memory access state (or any state before it if stalling at the access state causes a deadlock), and this logic queries all structures that can hold M1 in a pre-completion state.
- Adds logic to the structure holding M2 to stall M2 until receiving a subsequent un-stall message from a structure holding M1 in a pre-completion state.
- Adds logic to all structures that can hold M1 in a pre-completion state, and this logic sends an un-stall message to the structure holding M2 when M1 changes its instruction state to its completion state.

This stall we add is conservative in that not all controllers that can hold M1 in a pre-complete state need to be queried. For example, in Design-1, a load is inserted in the IQ, LQ, and ROB, and has state in all of these controllers. Querying all of these controllers' state is redundant, because the LQ holds all in-flight loads and their instruction states. The IQ and ROB do not contribute any information about loads beyond what the LQ already has.

4.2 Load-Replay

The load replay transformation has the pipeline replay each load at ReadyToCommit (reusing the originally computed address) and compare the replayed value to the value that was returned from the memory system when the load earlier executed speculatively. To implement this transformation, PipeGen must:

- Add logic to re-issue each load to the memory system when the load is ready to commit.
- Identify where the original, speculative value is held.
- Add logic to compare the replayed load value to the original, speculative value. If the values differ, the logic must use the misspeculation recovery logic provided by the user for recovering from single-core misspeculation.

4.2.1 Adding Logic to Replay Loads. PipeGen must first identify where in the pipeline loads are issued to the memory system. Because the user labels the canonical instruction states, PipeGen can find where a load transitions from ReadyToIssue to WaitingForResponse; this transition occurs in the structure that issues loads. We refer to this structure as the load-issuing structure. PipeGen adds logic to the load-issuing structure to make it perform the replay load. If the system contains a structure that holds committed writes before they are written to memory, such as a post-commit write buffer (WB), that structure is logically part of the memory system and it is queried at replay. (PipeGen is able to identify whether or not an input pipeline has a post-commit WB by performing reachability analysis on post-commit store states.)

To identify whether there is a post-commit WB, once again PipeGen extracts the instruction state graph—this time of a store instruction. PipeGen then performs a reachability analysis starting from the store's ReadyToCommit state, and identifies the set of post-commit states of the store. Within the post-commit states of the store, PipeGen searches for the ReadyToIssue and WaitingForResponse states, and a match indicates the presence of a post-commit WB.

PipeGen must similarly identify where in the pipeline that loads commit, which we refer to as the commit structure. At that location, PipeGen adds logic to send a message from the commit structure to the load-issuing structure, requesting a load to be issued and stalling until the response from the memory system arrives at the load-issuing structure.

4.2.2 Identifying Where the Original Load Value Is. PipeGen searches for the sub-operation keyword write_load_result. The structure that includes that keyword is the structure that holds the data that has been returned by the memory system.

4.2.3 Adding Logic to Compare the Original Value to the Replayed Value. PipeGen identifies the structure where a load commits, using the commit sub-operation keyword. PipeGen adds logic there to (a) send a message to the structure that holds the originally read value of the load, requesting a response with the original value, (b) send a message to the load-issuing structure to replay the load and receive a newly read value, (c) compare the two values, and (d) if the values differ, trigger the single-core misspeculation recovery mechanism.

4.3 Invalidation Tracking

The invalidation tracking transformation observes incoming coherence invalidations and compares their addresses to the addresses of in-flight loads that have already speculatively executed. To implement this transformation, PipeGen makes the following transformation:

- Adds a structure to track the addresses of speculative loads.
- Adds logic to compare the addresses of speculative loads to the addresses of incoming coherence invalidations.

4.3.1 Adding Structure to Track Addresses of Speculative Loads. PipeGen adds a structure whose entries are the addresses (and sequence numbers) of loads that have already been speculatively executed. The entries of this structure, which we refer to as LoadTracker, are unordered. LoadTracker can be queried by address, and it accepts messages for adding and removing entries. PipeGen adds logic to (a) add an entry whenever a load performs the sub-operation issue_load_request and (b) remove an entry whenever a load performs the sub-operation commit. LoadTracker can also accept messages to reset it (i.e., clear all of its entries).

4.3.2 Adding Logic to Compare Addresses of Speculative Loads to Invalidations. PipeGen adds logic to the memory system to send the addresses of incoming coherence invalidations to LoadTracker. If there is a match, LoadTracker resets itself and triggers the pipeline's single-core misspeculation mechanism to squash the misspeculated load and all instructions younger than it.

5 Usage Model

In the last section we discussed the three transformations used for enforcing a given MCM. How does the architect specify what transformations to employ? More generally, what is the usage model of PipeGen?

Conceptually, PipeGen takes as input a single-threaded pipeline and the MCM and produces a pipeline that enforces the MCM. More concretely, the input pipeline is expressed in AQL, our DSL. What about the MCM? The MCM is expressed as a MOST table [18, 19] which is a standardized format for expressing the memory orderings enforced by the MCM. For example, the MOST table for x86TSO specifies that TSO enforces the Load → Load, Load → Store, and the Store → Store orderings, and the fact that an x86TSO fence instruction, mfence, is ordered with loads, stores, and other mfences.

PipeGen first employs extensive litmus testing to determine what orderings are already enforced by the input pipeline. For example, in most input pipelines, the Load → Store ordering is enforced by default because stores are only allowed to issue to the memory system after any previous loads commit. For the orderings that are not enforced by default, PipeGen must enforce them. But the architect must specify which transformation to use for enforcing each ordering.

We introduced three transformations in the previous section: inorder (IO), load replay (LR), and invalidation tracking (IT). As we discussed in the previous section, IO is the most flexible and can enforce all 4 combinations of memory orderings and fences. LR and IT can enforce the M → Load ordering (where M is a load, store, or fence). Therefore, for every ordering in the MOST table, PipeGen could potentially offer a choice if that ordering can be enforced by both IO and LR/IT. The architect should then specify what transformation must be employed for enforcing that ordering guarantee. We leave this to the architect because they are in the best position to make this choice depending on the type of pipeline. As an extreme example, consider an input pipeline that is not speculative and uses a post-commit write buffer which can violate the Store → Load ordering. In such a pipeline it makes sense to use only the IO transformation because of the lack of speculative capabilities in the original pipeline. We show the MOST table for the x86TSO processor in Table 3. As we can see, there are three choices for enforcing the Load → Load ordering; the architect chooses one of them. The Load → Store ordering happens to be already enforced and is indicated in the table. Other entries follow a similar pattern.

6 Output: Results and Verification

PipeGen's output is the transformed pipeline with the MCM mechanisms in place. Currently, PipeGen produces this output both in AQL and in the language of the Murphi model checker [9]. Other output formats are possible, such as Verilog or BlueSpec, but given that the first goal of PipeGen is ensuring functionality, Murphi is more useful.

Table 3: MOST table of TSO ordering transformation choices.

	Load	Store	mFence
Load	LR, IT or IO	Y	IO
Store	LR, IT or IO	IO	IO
mFence	LR, IT or IO	IO	IO

We have used the Murphi output to verify that PipeGen produces pipelines that correctly enforce the specified MCMs. We have explored combinations of three different single-core pipeline designs (Designs 1, 2, and 3), two different MCMs (x86TSO and ARMv8), and three MCM enforcement mechanisms (i.e., the three transformations from Section 4).

But before discussing the verification results, we show sample output generated by PipeGen to convey a feel for the automation performed.

6.1 Example Output generated by PipeGen

In this section, we show sample output generated by PipeGen for Design-3. Recall that in Design-3 loads can issue to the memory system out-of-order from the load buffer (LB), violating the Load → Load ordering. (We consider a design which has precisely one entry in the LB.) We use PipeGen to enforce this ordering using the In-order (IO) transformation.

At first look, manual IO enforcement might look trivial for this situation: stall the issue of a load to the memory system until all earlier (in program order) loads complete. An architect manually implementing this transformation might be tempted to implement this at the LB since that is where loads are issued to the memory system. But as discussed in Section 4.1 this could lead to a deadlock.

Why? Let us consider two loads, ld1 and ld2, with ld1 before ld2 in program order. Suppose ld2's address resolves first and it enters the LB. Now, if the IO enforcement were implemented at the LB, then the issue of ld2 would have been stalled until the earlier ld1 has its result from the memory system. But ld1 will never be able to enter the LB as that space is occupied by ld2.

This example illustrates that even a seemingly simple enforcement method like IO can be tricky when specializing for different pipelines. As discussed earlier, PipeGen identifies that loads do not enter the LB in program order and therefore searches "backwards" to identify a suitable structure where IO can be safely implemented. In Design-3, that structure is the issue queue (IQ).

Listing 2 shows the "before" and "after" versions of the issue queue for a load instruction (the "after" parts are shown in blue). As can be seen, the original pipeline code simply dispatches the load to the LB when the LB unit is ready (lines 22-28). PipeGen identifies the IQ as a suitable structure to enforce IO. It then adds the code to check whether there are undispatched (to the LB) prior loads (lines 3-17). This is a simple associative search within the entries of the IQ as shown. We argue that with the knowledge of this output snippet, it should be a fairly easy task for the architect to implement these changes in any hardware design language such as Verilog.

```
state IQScheduleInst{
  // ASSOCIATIVE SEARCH
  bool any_unexecuted_older_loads = false;
  await IQ.search((entry.instruction.seq_num <
                instruction.seq_num) &
                entry.instruction.op == ld,
                min(instruction.seq_num -
                    entry.instruction.seq_num)) {
    when search_success() from IQ {
      if (entry.curr_state == iq_schedule_inst) {
        any_unexecuted_older_loads = true;
      } else {
        any_unexecuted_older_loads = false;
      }
    }
    when search_fail() from IQ { }
  }
  // STALL IF THERE ARE ANY OLDER LOADS IN THE IQ
  if (any_unexecuted_older_loads) {
      reset IQScheduleInst;
  }
  if (instruction.op == ld) {
    if (load_buffer.state == ready) {
      load_buffer.execute_load(instruction);
      remove();
      complete IQWaitForInst;
    }
  }
}
```

Listing 2: Example output AQL code of the IQ in Design 3.

In summary, this example illustrates that: (1) Even seemingly simple transformations are not easy to specialize for a given pipeline; (2) PipeGen's automatic transformation solves this problem; and (3) It is relatively straightforward for an architect to look at the PipeGen output to implement the changes in any HDL.

6.2 Verification Methodology

We run litmus tests to check if all orderings allowed by the MCM are possible and all orderings disallowed by the MCM are impossible. By using Murphi to "run" the litmus tests we ensure that each litmus test is explored exhaustively (i.e., every possible interleaving of instructions is tested). While litmus testing is not a complete proof, it is widely used and quite effective.

The litmus tests we run cover multi-core orderings, checking that the algorithms have added the orderings required by x86TSO and ARMv8. The basic litmus tests are MP (Message Passing), Dekker's, LB (Load Buffering), and n7, where we additionally run several barrier variants of MP, LB, and Dekker's to test barrier instruction orderings required by x86TSO and ARMv8. A barrier instruction may be between instructions in the Dekker's, LB, and MP litmus tests, where stores and loads may be releases and acquires.

MP litmus test. The message passing (MP) litmus test is shown in Table 4. In MP, with standard stores and loads, Core 1's reads should not be able to read 1 and 0 in r1 and r2, respectively, if Store → Store and Load → Load are enforced.

Dekker's litmus test. The Dekker's litmus test is shown in Table 5. If Store → Load is enforced, then reading the results 0 and 0 in both cores is not possible, but it is possible if that ordering is relaxed.

Table 4: MP Litmus test, parameterized. With standard stores and loads, core 1 reading 1 and 0 in r1 and r2 respectively is not allowed in x86TSO, and allowed in ARMv8.

Core 0	Core 1
Store [x] 1	Load r1 [y]
[Opt. Barrier Inst.]	[Opt. Barrier Inst.]
Store [y] 1	Load r2 [x]

Table 5: Dekker's litmus test, parameterized. Core 0 and 1 both reading 0 is only possible if Store → Load is not enforced. This outcome is observable in x86TSO and ARMv8.

Core 0	Core 1
Store [x] 1	Store [y] 1
[Opt. Barrier Inst.]	[Opt. Barrier Inst.]
Load r1 [y]	Load r1 [x]

Table 6: The LB litmus test. If Cores 0 and 1 both read 1 in r1, then the MCM relaxes Load → Store. This outcome is allowed in ARMv8.

Core 0	Core 1
Load r1 [x]	Load r1 [y]
[Opt. Barrier Inst.]	[Opt. Barrier Inst.]
Store [y] 1	Store [x] 1

Table 7: The n7 litmus test. If Cores 0 and 2 both read 1 and 0 in r1 and r2, then the MCM relaxes Store → Load or Load → Load. This outcome is allowed both in x86TSO and ARMv8.

Core 0	Core 1	Core 2
Store [x] 1	Store [y] 1	Load r1 [y]
Load r1 [x]		Load r2 [x]
Load r2 [y]		

LB litmus test. The load buffering LB litmus test is shown in Table 6. If Load → Store is enforced, then reading 1 in both cores is not possible, but is possible if the ordering is relaxed.

N7 litmus test. The n7 litmus test is in Table 7. In this test, a LSQ that has a post-commit WB and that enforces Load → Load, but does not enforce Store → Load, permits (a) Core 0 reading 1 into r1 and reading 0 into r2 and (b) Core 2 reading a 1 into r1 and 0 into r2. The second load in Core 0 thus reads from address y before Core 1's store writes to y, and after it writes to y, Core 2's first load reads from y. Then the second load reads 0 from x as Core 0's store is in its SB.

As TSO includes the mfence instruction, we exhaustively test all combinations of load and store instructions with an mfence added between the instructions, resulting in 7 litmus tests per design. Similarly, ARMv8 introduces LDAR, STLR, DMB SY, DMB ST, and

DMB LD, resulting in 22 litmus tests per design after adding litmus test variations of MP, LB, and Dekker's with these ARMv8 fence instructions.

6.3 Transformation Combinations Used in Experiments

PipeGen can apply three different transformations, described in Section 4, and it can apply them in isolation or in combinations. We consider three combinations:

- In-order memory instructions only. The in-order memory instruction transformation can be used to provide any ordering that is desired.
- In-order memory + load replay. This combination uses load replay to provide some orderings (M → Load) and in-order memory for the rest.
- In-order memory + invalidation tracking. This combination uses invalidation tracking to provide some orderings (M → Load) and in-order memory for the rest.

In Table 8, we show how we use these three transformation combinations for a cross-product of the three designs and two MCMs we consider in this evaluation. As there are 3 LSQ microarchitectures, and 3 combinations of transformations to evaluate on each microarchitecture for each MCM (TSO and ARMv8), we litmus test 9 experiments per MCM. Across TSO and ARMv8, we litmus test the 7 and 22 litmus tests respectively from Section 6.2 per experiment, for a total of 63 and 198 litmus tests.

6.4 Results for Design 1

Design-1 lets loads and stores execute in any order (that does not violate single-thread correctness), and as a result, it does not enforce Load → Load, Store → Store, or Store → Load orderings. Design-1 does enforce Load → Store though, as loads have completed their access when they commit, and stores have not yet been issued to the memory system when they commit.

6.4.1 Verifying Transformation Combinations that Enforce TSO. The results corroborate that all of the generated pipelines behave as expected, allowing the allowable litmus test outcomes and disallowing the prohibited outcomes. Unlike some other design/MCM pairs we discuss later, the transformations do not cause any overly conservative orderings.

6.4.2 Verifying Transformation Combinations that Enforce ARMv8. The results are shown in Table 9, where most expected behaviors are exhibited, except for the grey cells. The enforced MCM is slightly stronger than the ARMv8 MCM. Specifically, in the Dekker's litmus test with an added DMB ST fence, transformation combinations with Invalidation Tracking and Load Replay both enforce Store → DMB-ST → Load. This is because DMB-ST stalls until prior stores complete, and loads following the DMB-ST can only commit after the DMB-ST commits. With invalidation tracking and load replay taking care of misspeculated loads, this results in Store → DMB-ST → Load to be enforced.

6.5 Results for Design 2

Design-2 is similar to Design-1 in terms of the orderings it enforces, except that without a post-commit WB, stores are executed in-order at commit. Loads still execute out of order and forward from older stores. Thus Design-2 enforces Load → Store by default (like Design-1) but also Store → Store because of the lack of a WB.

6.5.1 Transformation Combinations to Enforce TSO. Design-2's single LSQ disallows the weaker orderings (in the yellow cells of Table 10) from the litmus tests when tested with the Load Replay and Invalidation Tracking transformations, even for Dekker's and n7, as Design-2 has no post-commit WB, resulting in Store → Load being enforced. (Different from grey cells, the yellow cells indicate that while the relevant orderings are disallowed by PipeGen, the orderings would also have been disallowed by any manual enforcement technique. Thus, yellow cells are disallowed not because of the conservatism demonstrated by PipeGen but because of the design itself.)

6.5.2 Algorithms Combinations to Enforce ARMv8. Shown in Table 11, Design-2 with only the in-order algorithm applied meets the expected orderings, except in the event of Dekker's with the DMB LD barrier (the grey cell) which behaves stronger than required; this is because the load is stalled until the barrier is committed which can't commit until the store completes. With invalidation tracking and load-replay the test results (the yellow cells) are all disallowed as there is no post-commit WB, causing the two mechanisms to add Load → Load and Store → Load, as loads' speculation is validated after older stores complete.

6.6 Results for Design 3

Although Design-3 is different from Design-2 (microarchitecturally speaking), like Design-2, it also enforces Store → Store and Load → Store by default (and violates the other two orderings) while also relaxing Store → Load for the same address. Therefore, the results that we observe for Design-3 are identical to those observed for Design-2 in enforcing x86TSO as well as ARMV8 (Tables 10 and 11), except for N7 that makes use of Store to Load forwarding in its allowed outcome.

6.7 Summary

In summary, our results show that PipeGen always generates pipelines that adhere to the intended MCM; i.e., our generated pipeline never violates the intended MCM. In a couple of examples (involving fences – the grey cells) PipeGen shows slightly stronger behavior than what the intended MCM would show.

7 Related Work

Verification. There has been a rich history of verifying processor pipelines using theorem proving and model checking [5, 12, 13]. Each of these techniques verify a model of the processor, typically expressed as a state machine, and verify it against the instruction set specification. More recently PipeCheck [17] uses the μspec representation of pipelines to verify the MCM orderings of a pipeline using exhaustive litmus testing. For each litmus test, PipeCheck constructs a happens-before graph of the memory instruction events, and a cycle in this graph corresponds to whether an outcome is

Table 8: Three tested ordering mechanism combinations per design. Each combination is either IO (In-order), IO + LR (using IO and Load Replay), and IO + IT (using IO + Load Replay). An ordering may already be enforced (Y) or unenforced (N). *Design 3 does not enforce Store to Load for the same address.

	Design 1, TSO			Design 2, TSO			Design 3, TSO		
TSO Orderings	IO	IO + LR	IO + IT	IO	IO + LR	IO + IT	IO	IO + LR	IO + IT
Ld → Ld	IO	LR	IT	IO	LR	IT	IO	LR	IT
Ld → St	Y	Y	Y	Y	Y	Y	Y	Y	Y
St → St	IO	IO	IO	Y	Y	Y	Y	Y	Y
St → Ld	N	N	N	N	LR	IT	N*	LR	IT
Ld → mF	Y	Y	Y	Y	Y	Y	Y	Y	Y
St → mF	IO	IO	IO	Y	Y	Y	Y	Y	Y
mF → Ld	IO	LR	IT	IO	LR	IO	IO	LR	IO
mF → St	Y	Y	Y	Y	Y	Y	Y	Y	Y
mF → mF	Y	Y	Y	Y	Y	Y	Y	Y	Y

	Design 1, ARM			Design 2, ARM			Design 3, ARM		
ARM Orderings	IO	IO + LR	IO + IT	IO	IO + LR	IO + IT	IO	IO + LR	IO + IT
Ld → Ld	N	LR	IT	N	LR	IT	N	LR	IT
Ld → St	Y	Y	Y	Y	Y	Y	Y	Y	Y
St → St	N	N	N	Y	Y	Y	Y	Y	Y
St → Ld	N	N	N	N	LR	IT	N*	LR	IT
LDA → Ld, LDA	IO	LR	IT	IO	LR	IT	IO	LR	IT
LDA → St, STR	Y	Y	Y	Y	Y	Y	Y	Y	Y
Ld, LDA → STR	Y	Y	Y	Y	Y	Y	Y	Y	Y
St, STR → STR	IO	IO	IO	Y	Y	Y	Y	Y	Y
Ld, LDA → DMB SY/ST/LD	Y	Y	Y	Y	Y	Y	Y	Y	Y
DMB SY/LD → Ld, LDA	IO	LR	IT	IO	LR	IT	IO	LR	IT
St, STR → DMB SY/ST	IO	IO	IO	IO	IO	IO	IO	IO	IO
DMB SY/ST/LD → St, STR	Y	Y	Y	Y	Y	Y	Y	Y	Y
DMB SY/ST/LD → DMB SY/ST/LD	Y	Y	Y	Y	Y	Y	Y	Y	Y

disallowed. This is similar to Herd [3], where cycles in memory instruction executions indicate a disallowed outcome. In contrast to pipeline verification, our work is focused on the top-down correct-by-construction [8] generation of pipelines, in which we add the required MCM orderings, rather than verifying that an existing pipeline correctly enforces MCM orderings.

Hardware Description Languages. Our work is related to hardware description languages such as Verilog, VHDL, Chisel [4], and Bluespec [22], in that we share the end goal of generating hardware designs. The point of departure of our work is that our approach is not general: our goal is geared towards generating processor pipelines as opposed to general hardware designs; in fact, our DSL is specialized to processor pipelines, and our transformations take advantage of this domain knowledge.

Microarchitecture Description Languages. Our work is most closely related to what we call microarchitecture description languages. These works raise the level of abstraction of microarchitecture design by using a DSL for expressing aspects of microarchitecture and using compiler technology to lower it to hardware. The earliest example of this is the Teapot language [7] for generating coherence protocols. Other early works raised the level of abstraction of processor pipelines for single-core processors [15, 23].

This area has had a recent resurgence. With open instruction sets and the advent of hardware startups, there is a demand for faster, more reliable and cheaper microarchitecture design. ProtoGen [24] proposes a DSL for synthesizing cache coherence protocols that are correct by construction, and they also use model checking to verify the synthesized protocols. PDL [25], proposes a pipeline description language and supports limited out-of-order execution. But PDL does not support multicore processors and as a consequence does not help enforce MCMs. To summarize, ours is the only approach to our knowledge that automates the generation of MCMs at the pipeline level.

8 Conclusion

One of the biggest challenges in designing an out-of-order pipeline is ensuring that it correctly supports a desired MCM. While MCM enforcement mechanisms have been previously developed, applying them manually is challenging and error-prone. We have developed PipeGen to automate that design challenge. Specifically, given an MCM and a single-core pipeline design that is MCM-oblivious, PipeGen outputs a pipeline that correctly supports the given MCM. We have verified PipeGen for a variety of single-core pipelines, MCMs, and MCM enforcement mechanisms.

Table 9: Design 1 ARMv8 litmus test results.

Design 1	ARMv8 Expected	In-Order	In-Order + Inval	In-Order + Load-Replay
MP	Allow	Allow	Allow	Allow
MP DMB_SY	Disallow	Disallow	Disallow	Disallow
MP DMB_LD DMB_ST	Disallow	Disallow	Disallow	Disallow
MP DMB_LD DMB_ST Mismatch	Allow	Allow	Allow	Allow
MP Ordered LDAR STLR	Disallow	Disallow	Disallow	Disallow
MP Unordered LDAR STLR	Allow	Allow	Allow	Allow
MP Only LDAR STLR	Disallow	Disallow	Disallow	Disallow
Dekker's	Allow	Allow	Allow	Allow
Dekker's LDAR	Allow	Allow	Allow	Allow
Dekker's STLR	Allow	Allow	Allow	Allow
Dekker's LDAR STLR	Allow	Allow	Allow	Allow
Dekker's DMB_SY	Disallow	Disallow	Disallow	Disallow
Dekker's DMB_LD	Allow	Allow	Allow	Allow
Dekker's DMB_ST	Allow	Allow	Disallow	Disallow
LB	Disallow	Disallow	Disallow	Disallow
LB LDAR	Disallow	Disallow	Disallow	Disallow
LB STLR	Disallow	Disallow	Disallow	Disallow
LB LDAR STLR	Disallow	Disallow	Disallow	Disallow
LB DMB_SY	Disallow	Disallow	Disallow	Disallow
LB DMB_ST	Disallow	Disallow	Disallow	Disallow
LB DMB_LD	Disallow	Disallow	Disallow	Disallow
n7	Allow	Allow	Allow	Allow

Table 10: Design-2 (Design-3) TSO litmus test results. *Design-3 disallows N7, as it doesn't forward from stores.

Design 2	x86TSO Expected	In-Order	In-Order + Inval. Handling	In-Order + Load Replay
Message Passing	Disallow	Disallow	Disallow	Disallow
MP Fence	Disallow	Disallow	Disallow	Disallow
Dekker's	Allow	Allow	Disallow	Disallow
Dekker's Fence	Disallow	Disallow	Disallow	Disallow
LB	Disallow	Disallow	Disallow	Disallow
LB Fence	Disallow	Disallow	Disallow	Disallow
n7	Allow	Allow*	Disallow	Disallow

Table 11: Design-2 (Design-3) ARMv8 litmus test results. *Design-3 doesn't allow N7, as it doesn't forward from stores.

Design 2	ARMv8 Expected	In-Order	In-Order + Inval	In-Order + Load-Replay
MP	Allow	Allow	Disallow	Disallow
MP DMB_SY	Disallow	Disallow	Disallow	Disallow
MP DMB_LD DMB_ST	Disallow	Disallow	Disallow	Disallow
MP DMB_LD DMB_ST Mismatch	Allow	Allow	Disallow	Disallow
MP Ordered LDAR STLR	Disallow	Disallow	Disallow	Disallow
MP Unordered LDAR STLR	Allow	Allow	Disallow	Disallow
MP Only LDARs STLRs	Disallow	Disallow	Disallow	Disallow
Dekker's LDAR	Allow	Allow	Disallow	Disallow
Dekker's STLR	Allow	Allow	Disallow	Disallow
Dekker's LDAR STLR	Allow	Allow	Disallow	Disallow
Dekker's	Allow	Allow	Disallow	Disallow
Dekker's DMB_SY	Disallow	Disallow	Disallow	Disallow
Dekker's DMB_LD	Allow	Disallow	Disallow	Disallow
Dekker's DMB_ST	Allow	Allow	Disallow	Disallow
LB	Disallow	Disallow	Disallow	Disallow
LB LDAR	Disallow	Disallow	Disallow	Disallow
LB STLR	Disallow	Disallow	Disallow	Disallow
LB LDAR STLR	Disallow	Disallow	Disallow	Disallow
LB DMB_SY	Disallow	Disallow	Disallow	Disallow
LB DMB_ST	Disallow	Disallow	Disallow	Disallow
LB DMB_LD	Disallow	Disallow	Disallow	Disallow
n7	Allow	Allow*	Disallow	Disallow

PipeGen's fundamental contribution is a set of transformations that codify three different methods of enforcing MCMs at the pipeline. Another important contribution is the identification of important states and sub-operations which are essential to perform these transformations.

But this work is only the first step. While we have used our own DSL for specifying the input, it would be interesting to explore whether existing HDLs can be extended to convey the important annotations that are required to implement our transformations. Secondly, whereas a Murphi backend validates that PipeGen works correctly, it would be interesting to create an HDL backend.

Acknowledgments

This work is supported by the National Science Foundation under grant CCF-200-2737 and the Engineering and Physical Sciences Research Council, through grant EP/V038699/1.

References

[1] 2014. Gem5 TSO implemention which permits only one operation in the store buffer. https://gem5-users.gem5.narkive.com/RqRv5GVj/lsq-bottleneck-when-using-x86-tso

[2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running tests against hardware. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 41–44.

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 2 (2014), 1–74.

[4] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun (Eds.). ACM, 1216–1225. https://doi.org/10.1145/2228360.2228584

[5] Jerry R. Burch and David L. Dill. 1994. Automatic verification of Pipelined Microprocessor Control. In *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 818)*, David L. Dill (Ed.). Springer, 68–80. https://doi.org/10.1007/3-540-58179-0_44

[6] Harold W Cain and Mikko H Lipasti. 2004. Memory ordering: A value-based approach. *ACM SIGARCH Computer Architecture News* 32, 2 (2004), 90.

[7] Satish Chandra, Brad Richards, and James R. Larus. 1996. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 237–248. https://doi.org/10.1145/231379.231430

[8] Edsger W. Dijkstra. 1967. A constructive approach to the problem of program correctness. (Aug. 1967). http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF circulated privately.

[9] David L Dill. 1996. The Mur ϕ verification system. In *Computer Aided Verification: 8th International Conference, CAV'96 New Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8*. Springer, 390–393.

[10] Marco Elver and Vijay Nagarajan. 2016. McVerSi: A test generation framework for fast memory consistency verification in simulation. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 618–630. https://doi.org/10.1109/HPCA.2016.7446099

[11] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. 1991. *Two techniques to enhance the performance of memory consistency models*. Computer Systems Laboratory, Stanford University.

[12] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam K. Srivas. 2000. Verifying Advanced Microarchitectures that Support Speculation and Exceptions. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1855)*, E. Allen Emerson and A. Prasad Sistla (Eds.). Springer, 521–537. https://doi.org/10.1007/10722167_39

[13] Ravi Hosabettu, Mandayam K. Srivas, and Ganesh Gopalakrishnan. 1998. Decomposing the Proof of Correctness of pipelined Microprocessors. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1427)*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer, 122–134. https://doi.org/10.1007/BFB0028739

[14] Yao Hsiao, Dominic P Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. 2021. Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 679–694.

[15] Daniel Kroening and Wolfgang J. Paul. 2001. Automated Pipeline Design. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 810–815. https://doi.org/10.1145/378239.379071

[16] George Kurian, Omer Khan, and Srinivas Devadas. 2013. The locality-aware adaptive cache coherence protocol. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 523–534.

[17] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2014. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 635–646.

[18] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: defending against memory consistency model mismatches in heterogeneous architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, Deborah T. Marr and David H. Albonesi (Eds.). ACM, 388–400. https://doi.org/10.1145/2749469.2750378

[19] Daniel Joseph Lustig. 2015. *Specifying, Verifying, and Translating Between Memory Consistency Models*. Ph. D. Dissertation. Princeton University.

[20] Milo MK Martin, Daniel J Sorin, Harold W Cain, Mark D Hill, and Mikko H Lipasti. 2001. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 328–337.

[21] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04*. IEEE, 69–70.

[22] Rishiyur S. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*. IEEE Computer Society, 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[23] Eriko Nurvitadhi, James C. Hoe, Timothy Kam, and Shih-Lien Lu. 2011. Automatic Pipelining From Transactional Datapath Specifications. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 30, 3 (2011), 441–454. https://doi.org/10.1109/TCAD.2010.2088950

[24] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. 2018. ProtoGen: Automatically generating directory cache coherence protocols from atomic specifications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 247–260.

[25] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C Myers. 2022. PDL: a high-level hardware design language for pipelined processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 719–732.

A Artifact Appendix

A.1 Abstract

This paper provides an artifact, which includes a docker image, prepared with all the necessary files and an environment to reproduce the tables in the paper. The minimum requirements to run our artifact are 15 GB of RAM, and 10 GB of disk space. However, the more cores and RAM that's available, the faster the artifact evaluation will be, as we provide an automated script in the docker image to run experiments in parallel, and each experiment requires at least 15 GB of RAM.

The docker image contains a python3.10 script that will run all the experiments and corresponding litmus tests to reproduce the tables in the paper. The instructions to run the script are in the `README-Artifact-Evaluation.md` README file in the docker image.

A.2 Artifact check-list (meta-information)

- **Run-time environment: Docker image.**
- **Hardware: Use a computer with enough RAM for the experiments, at least 15 GB. The more cores and memory, the better, as each litmus test will use the 'at least 15 GB', and multiple litmus tests can be run in parallel.**
- **Execution: Open docker image, run automated python test script to run all experiments and corresponding litmus tests.**
- **Output: Result of litmus tests, shows if MCM orderings are enforced. Check that the output matches the paper's expected results.**
- **Experiments: Use our PipeGen framework to transform LSQs, run litmus tests in the Murphi model checker.**
- **How much disk space required (approximately)?: 10 GB**
- **How much time is needed to prepare workflow (approximately)?: 10 minutes (docker)**
- **How much time is needed to complete experiments (approximately)?: 4-5 days**
- **Publicly available?: Yes, on Zenodo**
- **Code licenses (if publicly available)?: Yes, MIT**

- **Data licenses (if publicly available)?: N/A**
- **Workflow framework used?: Python Script**
- **Archived (provide DOI)?: 10.5281/zenodo.12682811**

A.3 Description

A.3.1 How to access. Our artifact is publicly available at https://zenodo.org/uploads/12682811. The `artifact.tar` file on Zenodo is a tarball of a docker image. Please load the docker image (i.e. `docker load < artifact.tar`) and run it interactively (i.e. run the image with `docker run -it aql`). The docker container will have the artifact files, python script, and the `README-Artifact-Evaluation.md` file.

A.3.2 Hardware dependencies. There are no special hardware dependencies, the artifact should run on any modern computer.

A.3.3 Software dependencies. All software dependencies are contained already in the docker image. For completeness, the docker image uses: python3.10, pandas, tabular, Lean v4.9 (latest version), elan 3.1.1, cmurphi5.5.0, and g++.

A.3.4 Data sets. Not applicable.

A.3.5 Models. Not applicable.

A.4 Installation

Download the `archive.tar` file from Zenodo (https://zenodo.org/uploads/12682811). Run `docker load < archive.tar`, which will add the docker image `aql`. Run the docker image interactively with `docker run -it aql`.

A.5 Experiment workflow

Follow the instructions in the `README-Artifact-Evaluation.md` README file in the docker image. These are instructions for running the python script “`run-litmus-on-lsqs.py`" to reproduce the experiments and tables in the paper. Line 356 in the python script makes the script run all experiments. Line 358 in the script shows an example of running specific experiments. The workflow this script automates consists of applying transformations on input LSQ microarchitectures per memory model and main transformation experiment from the paper. Each of these experiments is evaluated with litmus tests in a model checker, to examine the resulting memory model orderings the produced microarchitecture enforces. These memory model orderings are collected in tables, as shown in this paper.

A.6 Evaluation and expected results

As described above, this artifact transforms a LSQ microarchitecture to enforce a specified memory model. The evaluation of whether a memory model is enforced, and determining which orderings are enforced as a result of PipeGen’s transformations is performed by checking litmus tests in the Murphi model checker. Litmus tests check for at least one memory model ordering, and as a memory consistency model consists of a number of orderings, by checking the enforced orderings with litmus tests, we can determine if a microarchitecture enforces a memory consistency model. If a microarchitecture enforces a few more orderings than specified, this is conservative, but acceptable.

The results of the litmus tests are printed in tables, and should match the paper’s table. The one exception being the LB LSQ microarchitecture with the N7 litmus test, the ARM memory model, enforced with only the IO (In-Order) transformation, which is disallowed, and will be updated and explained in the camera-ready version.

A.7 Experiment customization

Experiments can be customized if desired, by changing the transformations used.

A.8 Notes

This artifact uses a docker image tarball, containing all dependencies, and an automated script to run all experiments in the paper.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html