PipeGen: Automatically Transforming a Single-Core Pipeline into a Multi-core Pipeline Enforcing a given Memory Model

An Qi Zhang
University of Utah
USA

Andrés Goens
University of Amsterdam
The Netherlands

Nicolai Oswald
Nvidia
USA

Tobias Grosser
University of Cambridge
UK

Daniel Sorin
Duke University
USA

Vijay Nagarajan
University of Utah
USA

ABSTRACT

Designing a pipeline for a processor core is difficult. One major challenge is designing it such that it can be composed with other cores, while correctly enforcing the intended memory consistency model (MCM). Our goal is 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. This paper discusses the opportunity and challenges involved and our current progress in achieving this goal.

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 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 [8], or more worryingly, make a design error that leads to the desired MCM no longer being enforced [5, 6].

In this paper, we advocate for addressing this design challenge with automation. Specifically, we propose that an architect should be able to design a pipeline in an MCM-oblivious fashion, needing only to enforce single-threaded correctness. The automation should then transform that pipeline into a multi-core pipeline that enforces the specified MCM (Figure 1). A key insight is that we can achieve this automation as *code transformations* in a domain-specific language (DSL), borrowing methods from compiler design.

We now discuss each aspect of the desired flow—inputs, transformation algorithms, and outputs—including our current solutions and open questions.

2 INPUTS

Given an MCM-oblivious pipeline, a transformation algorithm must be able to identify the key microarchitectural sub-operations of a memory operation. (Memory operations are loads, stores, read-modify-writes, and fences). For example, the sub-operations of a

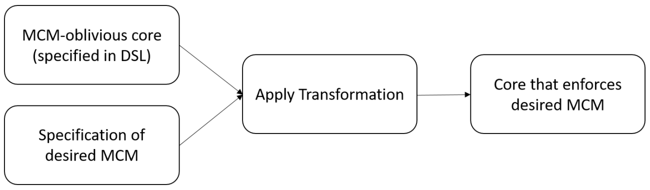
MCM-oblivious core (specified in DSL)
Specification of desired MCM
Apply Transformation
Core that enforces desired MCM

Figure 1: Automation goal: Given a pipeline in our DSL that correctly enforces single-threaded correctness, and a desired MCM, we automatically enforce the required MCM via three transformations that codify three different ways of enforcing memory ordering at the pipeline

load in a high-performance core include: being dispatched to the ROB and LSQ, being issued to the memory system once its address has been computed, speculatively writing to the register file the result it receives from the memory system, and being committed. Why do we need to identify the sub-operations? At its heart, enforcing memory ordering at the pipeline is all about controlling when and how these sub-operations occur.

In theory, an architect could specify the core pipeline using any language for expressing finite state machines. Verilog, BlueSpec, and Murphi [2], for example, would all suffice. However, none of these general-purpose languages would make it easy for an automated tool to identify the sub-operations that matter for enforcing MCMs. 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.

We have developed a DSL that enables the architect to model the core pipeline as an interconnected group of structures that each have some number of entries. These structures include the Reorder buffer (ROB), Load-store-queue (LSQ), instruction queue, etc. Each entry is logically a state machine; the entry has a state, and an event can cause an action and possibly a transition of that entry to new state. Importantly, the DSL also enables the architect to label where sub-operations occur. Our DSL enables relatively high-level specifications that focus on functionality more than cycle-accurate behavior.

Our DSL suffices for the automation we are doing, but questions remain. It is not clear that users will find a DSL preferable to a version of an already established language. We also do not know whether our DSL suffices for any type of core. We have explored a variety of cores (e.g., with different structures for speculative load execution) and a variety of mechanisms for transforming cores, but that does not guarantee that all possible core types or transformations will be expressible with our current DSL. Lastly, one can imagine specifying cores at varying degrees of detail; our current DSL is high-level, which may or may not suffice for certain purposes.

3 ALGORITHMS

Akin to a compiler, the core of our automation flow consists of algorithms for analyzing the input design and transforming it to enforce the desired MCM.

3.1 Analysis

The DSL, in both its format and its labeling features, facilitates automated analysis of the flow of memory operations through the core. The analysis algorithms can determine which core structures each operation passes through and what sub-operations occur in which structures. Furthermore, when a memory operation resides in a structure (e.g., LSQ), it has a state that can change due to a sub-operation, and the analysis algorithms can determine the sequence of states that a memory operation will traverse during its lifetime.

For example, for a "typical" out-of-order core, the analysis algorithms might determine that a load begins in the instruction queue (IQ), is dispatched into the ROB and LSQ, is issued to the memory system from the LSQ, has its result written into the LSQ (changing its entry's state in the LSQ), and then is committed by removing it from the LSQ and ROB.

3.2 Transformation

The transformation algorithms use the results of the analysis algorithms to determine what to do. We consider the transformations to effectively be compiler passes that perform source-to-source transformation. Currently, we have three types of transformations codifying three ways of enforcing memory ordering: adding stalls in the pipeline for enforcing ordering, detecting MCM violations via coherence tracking [3], and detecting MCM violations via load replays [3]. Given an MCM and a pipeline enforcing single-threaded correctness, these transformations can be used to ensure that the resulting output pipeline maintains that MCM.

Stall transformations are the most intuitive, in that they prevent something from happening that could violate the desired MCM. For example, an MCM could prohibit load-to-load reordering to different addresses, which is legal for an MCM-oblivious core. The stall transformation could stall a load from issuing to the memory system until its predecessor load (in program order) has completed, thus eliminating the possibility of those loads appearing out of order. Implementing a stall simply requires knowing at what point an operation becomes visible beyond the core.

The coherence tracking transformation adds a structure to the MCM-oblivious core to record the addresses of speculative loads that have been issued but not committed. This load address tracking structure is connected to the memory system such that it can receive coherence requests that invalidate blocks. If a coherence request invalidates a block that is in the address tracking structure, the corresponding load must be considered misspeculated in certain MCMs. For this transformation to be viable, we must have a mechanism for squashing a given instruction and its successors in program order. Our DSL enables the architect to specify how to trigger misspeculation logic; it is the same logic as that used for recovering the MCM-oblivious core from (single-thread) misspeculation.

The load replay transformation adds logic to the MCM-oblivious core to re-issue each load to the memory system at commit time and compare the values of the two loads. If they differ, the loads could have been visible out of order, and thus any instructions after the load must be considered misspeculated in an MCM that prohibits load-to-load reordering. As with the previous transformation, it must use the MCM-oblivious logic for squashing to recover from misspeculation.

We have not exhausted the space of possible transformations. Our current set of transformations reflects how past architects have manually ensured their cores satisfy their MCMs, but it is possible that other transformations exist that do not have existing analogs. However, even if a tool's transformations just implement known MCM enforcement schemes, doing so automatically for a given pipeline offers a considerable reduction in complexity and opportunity for design bugs. (This is analogous to how compiler optimization passes are a win.)

4 OUTPUTS

The output of our automated tool is an MCM-aware core design, specified at the same level of abstraction as the MCM-oblivious core that was input to the tool. As with the input, it is a finite state machine that could be described in any of a variety of backends. Currently, our tool produces its output in the language of the Murphi model checker [2]. This output format is convenient, in that it enables us to model check our results and confirm that they are functionally correct. We can imagine providing other backends for the tool, such as Verilog and C++ code to plug into a simulator such as gem5 [7].

5 RESULTS

We have modeled three very different single-core out-of-order pipelines as inputs: the Hennessy and Patterson load-store-queue [4], the No-Store-Queue design [9], and the unified load/store queue [1]. Each of these single-core processors enforce only single-core correctness. We then applied our three transformation to enforce the TSO and the ARM memory consistency model. We then validated the generated Murphi models against several litmus tests in the model checker, and our results show that our generated designs enforced correctness while allowing for most of the weak behaviors permitted by these two memory models. Specifically, we ran 4 litmus tests against the generated multi-core-ready TSO pipelines (Message Passing, Dekker, N7, and Dekker with a mFence) validating that the three transformations correctly enforce TSO. Against the ARM multi-core-ready pipelines, we ran 11 litmus tests (Message Passing, Dekker, N7, and variations of Message Passing and Dekker with ARM's additional memory ordering load acquire, store release, and DMB instructions). Here, the DMB fences enforced by our transformations are stronger, as we enforce fences as two orderings, between the pre-fence instructions with the fence, and the fence and post-fence instructions.

REFERENCES

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

[2] David L. Dill. 1996. The Murφ Verification System. In *International Conference on Computer Aided Verification (CAV)*. 390–393.

[3] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing, ICPP '91, Austin, Texas, USA, August 1991. Volume I: Architecture/Hardware*. CRC Press, 355–364.

[4] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[5] 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.

[6] Doowon Lee. 2018. *Decompose and Conquer: Addressing Evasive Errors in Systems on Chip.* Ph. D. Dissertation.

[7] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *CoRR* abs/2007.03152 (2020). arXiv:2007.03152 https://arxiv.org/abs/2007.03152

[8] 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.

[9] Tingting Sha, Milo MK Martin, and Amir Roth. 2006. Nosq: Store-load communication without a store queue. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 285–296.