Insights from *Gen: Correct-by-construction Coherence Protocols

Vijay Nagarajan
University of Edinburgh
UK

Daniel Sorin
Duke University
USA

Nicolai Oswald
University of Edinburgh/NVIDIA
UK/Switzerland

ABSTRACT

The *Gen project[1] consists of a language for expressing atomic specifications of coherence protocols and a suite of tools for generating concurrent, hierarchical and heterogeneous coherence protocols. This position paper summarizes the insights and lessons learned from this project.

1 INTRODUCTION

Cache coherence protocols are notoriously hard to design and verify. If you don't believe us, simply google this phrase and you will find dozens of research papers repeating this sentiment. It is not just academics who find coherence difficult. Teams of professional designers with dedicated validation teams have produced chips with coherence bugs in them [1].

Coherence protocols can look misleadingly simple when one sees them first in a textbook. There are only three-to-five states and intuitive transitions between these states. What's so difficult? Three reasons: concurrency, hierarchy, and heterogeneity.

Textbook protocols assume that a transition from one stable state to another stable state is atomic, i.e., the transition happens or appears to happen instantaneously. Only a simplistic system model (e.g., cores connected to a simple atomic bus) can provide this atomicity, yet high-performance modern system architectures like multicore processors employ multi-hop interconnection networks and distributed directory protocols. Designing protocols to correctly handle this concurrency is challenging. The protocol complexity introduced by concurrency is revealed in the large number of possible *transient* coherence states, in addition to the handful of stable states.

Hierarchy is a time-tested design strategy for scalable systems, but it also greatly complicates the design and verification of the coherence protocol. There are more states, more transitions, and more possible concurrency.

Modern systems use a wide range of different general and specialized compute units resulting in increasingly heterogeneous architectures that share memory. Heterogeneity multiplies the design complexity, because communication patterns within a CPU, GPU or an accelerator are very different, often mandating bespoke coherence protocols for each case. To compose these very different protocols into a unified heterogeneous whole is hard. Recent academic work [6, 19] and industrial work [2–4] have developed interfaces that facilitate this process, but it is still quite challenging to manually compose the protocols.

2 *GEN

We address the challenge of designing highly-concurrent [20], hierarchical [21], and heterogeneous [22] cache coherence protocols by automatically generating them from atomic, stable state protocol (SSP) specifications. The input is expressed in our domain-specific-language for expressing atomic coherence protocols. If it is a hierarchical or a heterogeneous protocol, the input simply specifies the atomic protocol of each level or cluster. The *Gen tool automatically produces the concurrent, hierarchical or heterogeneous protocol, and in that process automatically generates all of the transient states and actions. The output produced is actually a state machine encoding of the protocol. We have a Murphi [14] backend so that the generated protocol can be verified for correctness. We are currently developing backends for SLICC [16] (for cycle accurate simulation) and Verilog.

We have a library of different protocols expressed in *Gen, and in particular *Gen can handle industrial protocols such as ARM's CHI protocol specification [8]. The project is starting to have impact. It has been used by a team from IIT Madras/Thales for their safety critical processor. Protocols generated by HeteroGen are being evaluated by a major company designing mobile SoCs. A team from NVIDIA Research is building upon HeteroGen for ongoing research and development. The ProtoGen [20][2], HieraGen [21][3] and HeteroGen [22][4] source code is available on GitHub.

3 INSIGHTS AND LESSONS LEARNED

We outline the insights and lessons learned from the *Gen project, with some caveats. First, *Gen is still research and has not been used extensively in products. Second, this is mainly about coherence protocols; whether or not these insights apply more generally to other aspects of microarchitecture design is an open question. Third, thus far the *Gen project generates state machines and not efficient hardware implementations.

3.1 Insights on the Interface

What is the input model? For any design automation or language-based microarchitecture project, it is crucial to ask this simple question: What is the input model? Ideally, the input representation is something that is easy for designers to express yet has enough detail for the designer to be able to control key aspects of the design. At the same time, it must be amenable to easy verification.

The stable state protocol (SSP) ticks all of these boxes for coherence protocols. Recall that this is a natural representation that textbooks often use to describe protocols. It frees up the protocol designer to think about the most important aspects of the protocols: what are its stable states and transitions, while completely relieving them from thinking about the concurrency aspects. Indeed, the protocol designer need not think about racing coherence transactions and transient states.

[1]This work is supported by Google, Huawei, EPSRC grants EP/V028154/1 and EP/V038699/1 to the University of Edinburgh, and the National Science Foundation under grant CCF-200-2737.

[2]https://github.com/Errare-humanum-est/ProtoGen.git
[3]https://github.com/Errare-humanum-est/HieraGen.git
[4]https://github.com/Errare-humanum-est/HeteroGen.git

In hindsight, the SSP appears to be an obvious input for coherence protocol automation tools. But there have been prior efforts at using a language-based approach for taming coherence – e.g., TRANSIT [23], Teapot [9, 10], Bluespec [12] – yet none of these approaches employ the SSP as the input representation[5].

Why? One potential reason is because of a pitfall that is easy to fall into. *Let the method decide the input instead of the other way around.* Consider for example TRANSIT [23], which applied counter-example-guided-inductive-synthesis (CEGIS) for coherence protocols. This method, however, can impact the input representation – and in this case concolic snippets of protocols, which we argue is not as natural for designers as the SSP.

That is why we believe this is an important takeaway: whatever cool new language and compiler you are developing for microarchitecture design, do you have the right input model? Are you solving the problem that matters?

The language. This should not come as a surprise, but once the input is precisely identified, it makes sense to specialize the language to the input representation. In case of cache coherence protocols, it is all about a bunch of cache controllers interacting with the directory controller to enforce cache coherence. In this case, the directory controller is different than the cache controllers and we choose to make this distinction explicit in the language. This distinction turned out to be critical to our transformation pass from the input to the output.

3.2 Insights on the Method

Cache coherence protocols, like other aspects of microarchitecture, have a clear correctness specification: the Single-Writer-Multiple-Reader (SWMR) invariant. Given the input and the correctness condition, how does one generate the output?

When not to employ Program Synthesis. It is quite tempting to use search-based techniques such as program synthesis [7] for these problems. Indeed, the problem seems to be tailor-made for program synthesis. Consider the input SSP to be the partial implementation with "holes", the transient states and actions being the holes; then, apply program synthesis to fill up the holes to match the specification.

However, we tried this approach but found that tools and solvers could not handle the large search space: even the modest MSI protocol required a search space larger than Avogadro's number. So we wrote our own synthesis engine that can exploit symmetry reduction and we implemented bespoke optimizations to reduce the search space [15]. Despite reducing the search space by over 99% we still could not fully synthesize the missing pieces of a simple MSI coherence protocol. The search space turned out to be too much!

A case for codifying domain expertise. Although cache coherence is hard, expert designers have been designing protocols for the past 30 years. Concurrency and hierarchy are hard problems, but there is a method to the madness: designers "know" how to tackle these problems. It is simply the case that these methods have not been formalized or codified.

The situation is similar to the early days of assembly programming – expert programmers knew how to optimize code, but these methods were yet to be codified into the now well-known compiler optimization techniques.

We argue that, at this nascent stage of language support for microarchitecture design, we should first understand, formalize and codify domain knowledge into compiler-like passes before embarking on search-based strategies. Let's expand on this by considering ProtoGen as a case study.

The input to ProtoGen is the SSP with stable states and atomic state transitions. ProtoGen produces a complete concurrent non-blocking protocol with all of the transient states and actions. This is a very hard problem in general: it requires a method for refining an atomic specification into a concurrent implementation. And this topic has received attention across a wide variety of areas ranging from databases to general hardware synthesis [5]. Yet, the existing general solutions either use some form of blocking or roll-back recovery. In ProtoGen we produce highly concurrent protocols without either of those. The key to this is exploiting domain knowledge. ProtoGen leverages the insight that, in a directory-based coherence protocol, racing transactions are serialized at the directory. By assigning a unique name to every directory-forwarded request that can arrive at a stable state in a cache, ProtoGen makes it possible for the directory to convey this serialization order to the caches. With the caches and the directory achieving consensus on the order of racing transactions, ProtoGen can generate highly-concurrent and non-blocking controller. Designers of coherence protocols have known this method intuitively, but it has not been codified thus far. ProtoGen codifies it.

The power of this design philosophy. This correct-by-construction approach [13] (i.e., refining a high-level abstraction down to the implementation while matching the specification) is not only helpful in automating "known" techniques, it is also helpful in thinking about new microarchitectural problems. *Specifically, it forces one to think carefully about the specification.* Often, the lack of a specification is what causes most correctness-related problems in microarchitecture design. Consider for example the history of memory models. Early memory models were implementation-driven: the implementation essentially decided the meaning of shared memory. We know that redressing this issue required 30 years of inter-disciplinary research across the semantics, PL and architecture communities.

History is about to repeat itself with heterogeneity. As heterogeneous processors are starting to share memory, the focus is again on implementations: coherence protocols such as CXL are starting to be get used for implementing heterogeneous shared memory. But what should be the specification? What should be the memory model when multiple devices share memory? In HeteroGen [22] we explicitly addressed this question and came up with an answer in the form of compound memory models, before going ahead with an implementation that honors the specification.

When to employ Program Synthesis? We do not believe Program Synthesis is useful when refining the abstraction level down from abstract to concrete. (Better to do this via compiler-like techniques.) However, we believe it can still be very useful for *searching sideways* for correct or efficient designs at the higher levels of abstraction, which have relatively smaller search spaces. Going back to the problem at hand, it might be useful for fixing bugs in the SSP [18] or searching across several different SSPs for performance.

[5]One notable exception is the conceptual framework of Nalumasu and Gopalakrishnan [17]. It is also worth noting that Hemiola [11], proposed after ProtoGen, uses an SSP-like input model.

REFERENCES

[1] [n. d.]. Coherency was broken and manually disabled in Galaxy S4. https://www.anandtech.com/show/7164/samsung-exynos-5-octa-5420-switches-back-to-arm-gpu. Accessed: 2022-05-05.

[2] [n. d.]. The AMBA CHI Specification. https://developer.arm.com/architectures/system-architectures/amba/amba-5. Accessed: 15th July 2019.

[3] [n. d.]. The GenZ Consortium. https://genzconsortium.org/. Accessed: 21st January 2019.

[4] [n. d.]. The OpenCAPI Consortium. https://opencapi.org/. Accessed: 21st January 2019.

[5] 2014. BluespecTM SystemVerilog Reference Guide. (2014).

[6] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 261–274. https://doi.org/10.1109/ISCA.2018.00031

[7] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*.

[8] Reece Carr. 2021. Developing a Library of Verified Cache Coherence Protocols. *4th Year Project Report* (2021). https://project-archive.inf.ed.ac.uk/ug4/20212191/ug4_proj.pdf

[9] Satish Chandra, Michael Dahlin, Bradley Richards, Randolph Y. Wang, Thomas E. Anderson, and James R. Larus. 1997. Experience with a Language for Writing Coherence Protocols. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997* (Santa Barbara, California) *(DSL'97)*. USENIX Association, USA, 5.

[10] S. Chandra, B. Richards, and J.R. Larus. 1999. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering* 25, 3 (1999), 317–333. https://doi.org/10.1109/32.798322

[11] Joonwon Choi, Adam Chlipala, and Arvind. 2022. Hemiola: A DSL and Verification Tools to Guide Design and Proof of Hierarchical Cache-Coherence Protocols. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13372)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 317–339. https://doi.org/10.1007/978-3-031-13188-2_16

[12] Nirav Dave, Man Cheuk Ng, and Arvind. 2005. Automatic synthesis of cache-coherence protocol processors using Bluespec. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), 11-14 July*. 25–34. https://doi.org/10.1109/MEMCOD.2005.1487887

[13] 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.

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

[15] Marco Elver, Christopher J. Banks, Paul Jackson, and Vijay Nagarajan. 2018. VerC3: A library for explicit state synthesis of concurrent systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1381–1386. https://doi.org/10.23919/DATE.2018.8342228

[16] 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

[17] Ratan Nalumasu and Ganesh Gopalakrishnan. 2002. Deriving Efficient Cache Coherence Protocols Through Refinement. *Formal Methods Syst. Des.* 20, 1 (2002), 107–125. https://doi.org/10.1023/A:1012916831123

[18] Theo Olausson. 2020. Towards the Automatic Synthesis of Cache Coherence Protocols. *4th Year Project Report* (2020). https://project-archive.inf.ed.ac.uk/ug4/20201774/ug4_proj.pdf

[19] Lena E. Olson, Mark D. Hill, and David A. Wood. 2017. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 163–176.

[20] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. 2018. ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. 247–260.

[21] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. 2020. HieraGen: Automated Generation of Concurrent, Hierarchical Cache Coherence Protocols. In *Procedings of the 47th Annual International Symposium on Computer Architecture*.

[22] Nicolai Oswald, Vijay Nagarajan, Daniel J. Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. 2022. HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 756–771. https://doi.org/10.1109/HPCA53966.2022.00061

[23] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174