Scalable Assurance via Verifiable Hardware-Software Contracts

Yao Hsiao Dominic P. Mulligan* Nikos Nikoleris* Gustavo Petri* Caroline Trippel
Stanford University *Arm Research
{yaohsiao, trippel}@stanford.edu {dominic.mulligan, nikos.nikoleris, gustavo.petri}@arm.com

I. Introduction

The correctness, reliability, and security of modern software depends on the hardware on which it is run. Thus, it is essential that hardware designers (1) expose relevant implementation details to software developers via *hardware-software contracts*, and (2) ensure, ideally through formal proof, that said contracts are indeed upheld by fabricated microarchitectures.

II. Background & Motivation

Hardware-Software Contracts: The canonical hardware-software contract is the instruction set architecture (ISA), which defines software-visible hardware state and a set of instructions which manipulate it. This simple notion of an ISA [2] served us well until the era of multi-core architectures. Notably, shared memory parallelism, combined with single-core optimizations that reorder and buffer instructions, gave rise to *memory consistency models* (MCMs) [18]—contracts which expose hardware ordering behaviors to software.

Another architecture trend exhibits challenges similar to those addressed by MCMs. Specifically, modern hardware is generally shared by many applications and contains an abundance of state that is read from and written to on these applications' behalf. This shared hardware state in combination with various data-dependent optimizations means that programs can interact and leak data that they process in unintended ways [39]. To address this issue, formal *security contracts* have gained traction among researchers as a way for hardware architects to expose security-critical implementation details—like those relevant for reasoning about microarchitectural leakage—to software [26, 25, 11, 14, 16, 15, 8, 38, 9].

The Hardware Assurance Challenge: Formal hardware-software contracts, including security contracts, are a promising way to encode assurance/certification requirements for hardware in a way that supports correct and secure software design. For example, a large body of work has produced formally specified MCMs for a variety of ISAs [35, 29, 1, 31, 28, 40] and high-level programming languages [23, 6, 30, 5, 3, 27], supporting the design of verified compilers [5, 4, 32, 20, 34, 30, 33, 37, 36]. Recent work also demonstrates that formal security contracts can support automated analysis tools which find [26, 15, 8, 10] and repair [26] microarchitectural leakage in programs. Unfortunately, despite their established benefits for software, a significant gap remains between existing formal contracts and the hardware designs they abstract.

Our goal is to devise techniques for synthesizing formal hardware-software contracts directly from hardware RTL.

III. Synthesizing Memory Model Specifications

A scalable, efficient, sound, and complete methodology for verifying processor MCM implementations has remained elusive due to modern design complexity. The closest approach, embodied in the *Check* tools [24], formally checks that a specific microarchitecture *in the guise of a manually constructed axiomatic specification*, called a μSPEC model, correctly implements an MCM.

Our recent work presents the only methodology and tool, RTL2μSPEC[1], which supports formally verifying the correctness of MCM implementations *down to RTL* [17]. Specifically, RTL2μSPEC enables the Check tools to consume processor RTL directly by automatically synthesizing μSPEC models from (System)Verilog implementations. We show that RTL2μSPEC can synthesize a complete, and proven correct, μSPEC model from the SystemVerilog design of the four-core ***open source RISC-V V-scale processor*** [21] in 6.84 minutes. Subsequent Check-based MCM verification of the synthesized μSPEC model takes on the order of seconds. Notably, prior work timed out out after 11 hours of runtime when attempting to verify the MCM of the same microarchitecture [22].

IV. Synthesizing Transmitter Specifications

Our current work extends RTL2μSPEC to synthesize *security contracts* from RTL. Simply put, our goal is to automatically discover the *transmit instructions* (or *transmitters*) [19, 41] of a microarchitecture. Transmitters are instructions which leak their results, operands, or even data at rest in hardware structures [39] via their variable usage of hardware resources (often via timing channels). We call our new methodology and tool TransmitSynth.

In adapting RTL2μSPEC to discover transmitters, we rely on the following observation: if an instruction is a transmitter it must be able to exhibit (observably) distinct *microarchitectural execution paths*. A microarchitectural execution path is a *set of state elements* that an instruction updates during its execution and a *partial order* on said state updates. For example, a load instruction might exhibit one execution path where it updates the data field of an L1 cache block (a cache miss) and another where it does not (a cache hit).

RTL2μSPEC's procedure for synthesizing μSPEC models from RTL lays the foundation for RLT transmitter discovery. In essence, μSPEC models are first-order logic (FOL) *ordering specifications* of a microarchitecture. They are composed of a set of *axioms* that describe how each legal hardware instruction

[1] RTL2μSPEC is open source at https://github.com/yaohsiaopid/rtl2uspec

(1) flows through the microarchitecture during its execution, with respect to which state elements it updates in which (partial) order (i.e., a microarchitectural execution path), and (2) interacts with other in-flight instructions during its execution via *structural* or *data-flow* dependencies. The *Check* tools use μSPEC models to reason about each possible execution of a progarm on a microarchitecture as a *directed acyclic graph*, called a μhb graph. Nodes in a μhb graph represent hardware events—namely, an instruction updating a particular state element during its execution; directed edges represent happens-before relationships between events. Fig. 1 shows three enhanced μhb graphs (featuring edge labels) which each represent an execution path of a single instruction.

RTL2μSPEC combines static analysis of an RTL netlist with SystemVerilog Assertion (SVA) property generation and verification. At a high level, it analyzes a netlist to synthesize SVAs corresponding to an over-approximation of all axioms that should be included in the final μSPEC model. Those which are proven—we use the JasperGold property verifier [7]—are retained and syntactically translated to the μSPEC DSL; those which are dis-proven (via counterexamples) are discarded.

While RTL2μSPEC can synthesize a microarchitectural execution path for an instruction, it faces a key limitation: *synthesizing multiple distinct execution paths for the same instruction is not supported.* RTL2μSPEC implicitly assumes that all instructions in the input design can only exhibit a single microarchitectural execution path (the *single execution path assumption* [17]). This restriction is fundamentally at odds with transmitter synthesis. Plus, it precludes analysis of processor designs with advanced features like speculation, out-of-order execution, bypassing, and multiple execution lanes.

Our TRANSMITSYNTH prototype removes RTL2μSPEC's single execution path assumption and successfully analyzes the `DIV` instruction on the ***open source RISC-V CVA6 processor core*** [42]—a 64-bit, 6-stage single issue RISC-V core with out-of-order write-back for each functional unit, *which is 12.6x larger than the V-Scale.*

V. TRANSMITSYNTH

We now describe TRANSMITSYNTH, which also combines static netlist analysis with SVA generation and verification.

Multiple Execution Graphs: For a given instruction under verification, TRANSMITSYNTH must synthesize the set of microarchitectural execution paths it may exhibit. To do so, it first determines which *Performing Locations* (PLs) the instruction may visit during its execution. A PL is a collection of state, which holds metadata corresponding to a single instruction (e.g., an instruction PC or other identifier) while the instruction "performs" state updates in a particular region of the design. In an in-order pipeline, PLs are pipeline stages.

At present, PLs are identified in the input design with the help of user-provided design metatdata. TRANSMITSYNTH then automatically derives which PLs an instruction may visit in any legal execution and the partial order on which they are visited. Fig. 1 shows three sets of PLs (denoted by each of the three columns) that may be visited by a RISC-V `DIV` instruction on CVA6. TRANSMITSYNTH generates two classes of specialized SVAs to derive these sets efficiently.

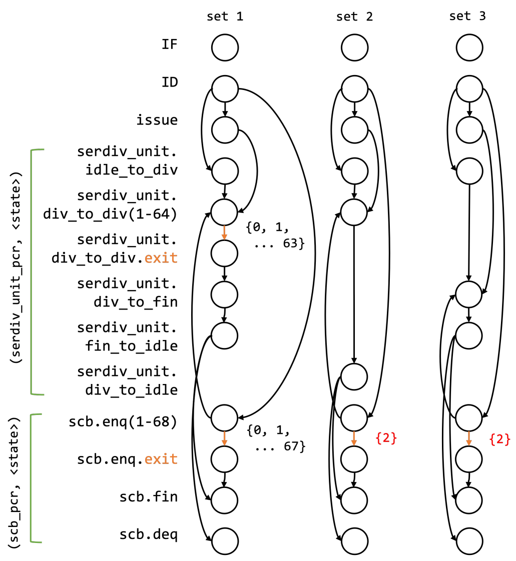

Figure 1. CVA6 `DIV` exhibits 66 execution paths

Happens-Before Latency: The different sets of PLs that an instruction can visit during its execution constitute one dimension of execution variability—one that manifests as distinct μhb graph nodes and edges. Another dimension results from the fact than an instruction may reside in a given PL for a variable number of cycles—one that we represent with a set of weights for μhb edges. An edge weight of `{0, 1, ..., 63}` in Fig. 1 indicates a 0, 1, ..., or 64 cycle happens-before latency. Edges without labels are implicitly labeled `{1}`. TRANSMITSYNTH leverages an iterative SVA generation and verification procedure to enumerate all possible wights for all μhb edges.

Characterizing Leakage: Transmitters may leak some function of: 1) their operands, 2) architectural data at rest, 3) microarchitectural data at rest, 4) structural resource contention. TRANSMITSYNTH's procedure for synthesizing distinct μhb graph structures (based on sets of visited PLs) can be restricted in order to coarsely categorize transmitters. For example, to identify transmitters which may leak a function of their operands and/or architectural data at rest, TRANSMITSYNTH can use a *non-interference assumption*, which requires that: (1) only one valid instruction, the instruction under verification, is issued after reset, and (2) architectural state (memory and registers) are initialized with symbolic values. We are extending TRANSMITSYNTH to supporting this broader space of transmitter categories.

Conclusions: When evaluating the CVA6 `DIV` instruction, under our non-interference assumption, TRANSMITSYNTH discovers *66 execution paths* (Fig. 1) in *96 minutes* of serial (but parallelizable) execution time. Three sets of PLs are identified, while one exhibits a variety of μhb edge latencies. ***TRANSMITSYNTH is the only tool [12, 13] that can enumerate a transmitter's execution paths.*** We plan to conduct fine-grained analysis of *what* transmitters leak in our next steps.

References

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.2 (2014), 7:1–7:74.

[2] Gene M. Amdahl, Gerrit A. Blaauw, and Frederick P. Brooks. "Architecture of the IBM System/360". In: *IBM Journal of Research and Development* (1964).

[3] Mark Batty, Alastair F. Donaldson, and John Wickerson. "Overhauling SC Atomics in C11 and OpenCL". In: *43rd Symposium on Principles of Programming Languages (POPL)* (2016).

[4] Mark Batty et al. "Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER". In: *39th Symposium on Principles of Programming Languages (POPL)* (2012).

[5] Mark Batty et al. "Mathematizing C++ Concurrency". In: *38th Symposium on Principles of Programming Languages (POPL)* (2011).

[6] Hans-J. Boehm and Sarita V. Adve. "Foundations of the C++ Concurrency Memory Model". In: *29th Conference on Programming Language Design and Implementation (PLDI)* (2008).

[7] Cadence Design Systems, Inc. *Cadence JasperGold formal verification platform*. Accessed 12$^{\text{th}}$ April 2021. URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.

[8] Sunjay Cauligi et al. "Constant-Time Foundations for the New Spectre Era". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020.

[9] Kevin Cheang et al. "A Formal Approach to Secure Speculation". In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 2019.

[10] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. "Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE". In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. 2021.

[11] Craig Disselkoen et al. "The Code That Never Ran: Modeling Attacks on Speculative Evaluation". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[12] Klaus v. Gleissenthall et al. "IODINE: Verifying Constant-Time Execution of Hardware". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019.

[13] Klaus v. Gleissenthall et al. "Solver-Aided Constant-Time Hardware Verification". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021.

[14] Roberto Guanciale, Musard Balliu, and Mads Dam. "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020.

[15] M. Guarnieri et al. "Spectector: Principled Detection of Speculative Information Flows". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020.

[16] Marco Guarnieri et al. "Hardware-Software Contracts for Secure Speculation". In: *2021 IEEE Symposium on Security and Privacy*. 2021.

[17] Yao Hsiao et al. "Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations". In: *Proceedings of the Fifty-Fourth IEEE/ACM International Symposium on Microarchitecture*. MICRO 54. 2021.

[18] IBM. *IBM System/370 Principles of Operation*. 1983.

[19] Vladimir Kiriansky et al. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018.

[20] Ori Lahav et al. "Repairing Sequential Consistency in C/C++11". In: *38th Conference on Programming Language Design and Implementation (PLDI)* (2017).

[21] Albert Magyar. *A Verilog implementation of the RISC-V Z-scale microprocessor*. https://github.com/ucb-bar/vscale. 2016.

[22] Yatin A. Manerkar et al. "RTLCheck: Verifying the Memory Consistency of RTL Designs". In: *Proceedings of the 50$^{\text{th}}$ International Symposium on Microarchitecture (MICRO)* (2017).

[23] Jeremy Manson, William Pugh, and Sarita Adve. "The Java Memory Model". In: *32nd Symposium on Principles of Programming Languages (POPL)* (2005).

[24] Margaret Martonosi et al. *Check: Research Tools and Papers*. http://check.cs.princeton.edu. 2017.

[25] Ross Mcilroy et al. *Spectre is here to stay: An analysis of side-channels and speculative execution*. 2019. URL: https://arxiv.org/abs/1902.05178.

[26] Nicholas Mosier et al. "Axiomatic Hardware-Software Contracts for Security". In: *ISCA'22*.

[27] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. "An Operational Semantics for C/C++11 Concurrency". In: *31st International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2016).

[28] NVIDIA. *Parallel Thread Execution ISA Version 6.0.* http://docs.nvidia.com/cuda/parallel-thread-execution/index.html. 2017.

[29] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: x86-TSO". In: *Proceedings of the 22$^{\text{nd}}$ International Conference on Theorem Proving in Higher Order Logics (TPHOLs)* (2009).

[30] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. "Cooking the books: Formalizing JMM implementation recipes". In: *29th European Conference on Object-Oriented Programming (ECOOP)* (2015).

[31] Christopher Pulte et al. "Simplifying Arm Concurrency: Multicopy-atomic Axiomatic and Operational Models for Armv8". In: *ACM Programming Languages* (2017).

[32] Susmit Sarkar et al. "Synchronising C/C++ and POWER". In: *33rd Conference on Programming Language Design and Implementation (PLDI)* (2012).

[33] Jaroslav Ševčık and David Aspinall. "On Validity of Program Transformations in the Java Memory Model". In: *Proceedings of the 22Nd European Conference on Object-Oriented Programming*. ECOOP '08 (2008).

[34] Peter Sewell. "C/C++11 mappings to processors". In: (2016). URL: https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

[35] Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence*. Ed. by Mark Hill. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2011.

[36] Viktor Vafeiadis and Chinmay Narayan. "Relaxed Separation Logic: A Program Logic for C11 Concurrency". In: *28th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2013).

[37] Viktor Vafeiadis et al. "Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It". In: *42nd Symposium on Principles of Programming Languages (POPL)* (2015).

[38] Marco Vassena et al. "Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade". In: *Proc. ACM Program. Lang.* (2021).

[39] Jose Vicarte et al. "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data". In: *ISCA'21*.

[40] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA Document, Version 20190608-Base-Ratified*. Tech. rep. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, June 2019. URL: https://riscv.org/specifications/.

[41] Jiyong Yu et al. "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019.

[42] Florian Zaruba and Luca Benini. "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640.