Research Summary

We entrust large parts of our daily lives to computer systems, and these systems require thorough verification to ensure their correctness. System vulnerabilities can lead to leakage of confidential information, loss of money, and (in the case of cyber-physical systems) may even threaten one's physical safety and security. Many such bugs have in fact been found in computing systems in recent years, ranging from security vulnerabilities (like Meltdown and Spectre [19]) to network errors [28] to concurrency bugs [8].

Some important bugs are due to flaws in system design/specification [8, 11], while others are due to implementations violating the design specification [28, 26]. Since verification is generally focused late in development (Figure 1a), bugs tend to get caught late in development as well. Design bugs may necessitate a redesign, wasting the development time spent creating incorrect implementations. Furthermore, many bugs only occur in very specific scenarios, making them hard to find. Current verification processes have proven inadequate for catching these bugs, resulting in the release of flawed hardware and software.

Verification using formal methods provides strong correctness guarantees based on mathematical proofs, and is adept at catching hard-to-find bugs. Formal verification may use automated [10] or manual approaches [32], but its use has historically been limited to the relatively few individuals with formal methods expertise, excluding typical hardware and software engineers.

I am a hardware and software systems researcher whose work lies on the boundary between systems and formal methods. My work aims to improve the verification of hardware and software using formal methods, without requiring engineers to have formal methods expertise. I propose a philosophy of Progressive Automated Formal Verification (Figure 1b), i.e. verification throughout the system development process, from early-stage design through to final implementation. Automating formal verification enables engineering teams to prove strong correctness properties about their designs and implementations by themselves, shifting the verification burden away from the relatively small number of formal methods experts.

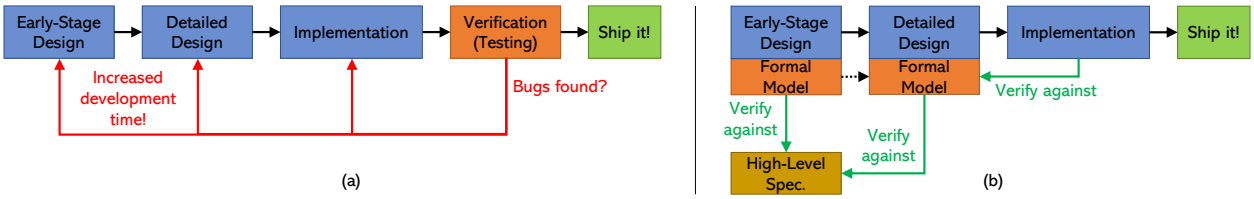

Figure 1: Traditional design flow (a) and a Progressive Automated Formal Verification flow (b).

In my philosophy, formal models of systems are first created and verified during early-stage design, thus ***catching design bugs before implementation commences***. These models can then *evolve* to become more detailed as system design progresses. Once the system is implemented, the implementation is formally verified against a proven-correct design model to help ensure implementation correctness. Amortizing verification over the entire design timeline in this manner can reduce verification overhead by finding bugs earlier. The number of steps in a progressive verification flow may vary depending on the type of system being built, but ***the key idea is to conduct verification throughout the design timeline and link the verification methods at each stage together***.

Inspired by this philosophy, ***my dissertation research has enabled progressive automated formal verification across the design timeline for memory consistency model (MCM) properties in parallel systems. My thesis work also stretches across the hardware/software stack***, from high-level programming languages down to low-level Verilog hardware circuits. In addition to enriching the systems community by enabling automated formal verification, ***my work has developed novel formal analysis techniques that benefit the formal methods community. My research has also had practical impact through the discovery of real-world bugs in C++ compilers, an open-source processor implementation (Verilog), and in the widely-used RISC-V instruction set (ISA).***

In the future, I will create methodologies, tools, and models for the ***principled design and verification of emerging heterogeneous parallel systems***. I will also make it easier for computer architects to design and verify hardware by ***raising the abstraction level of hardware description languages.*** Finally, I will develop techniques and tools that enable ***progressive automated formal verification in other domains***, such as distributed systems and cyber-physical systems. The long-term goal of my research is to make formal methods a ubiquitous technique for verifying computer systems.

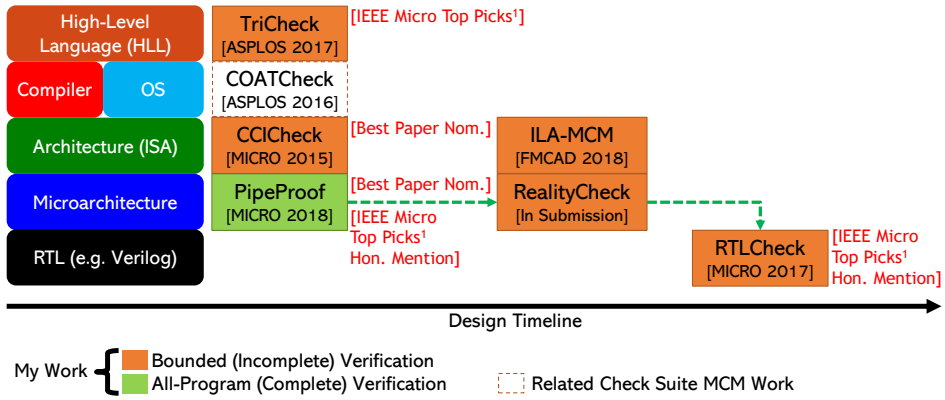

Figure 2: The hardware/software stack & design timeline, annotated with my MCM verification research.

Dissertation Research: Memory Consistency Model (MCM) Verification

Memory consistency models (MCMs) specify the ordering requirements for memory operations used for synchronization in parallel systems [30], so MCM verification is critical to parallel system correctness. MCMs are defined at various layers of the hardware/software stack (including high-level languages like C++ and ISAs like x86 or RISC-V), making MCM verification a full-stack problem. An MCM at a given layer of the stack (e.g. the ISA) defines the ordering guarantees that must be provided by lower layers of the stack (the hardware in this example), and consequently the guarantees that can be assumed by higher layers of the stack (in this example, the high-level language and compiler). If each layer of the stack does not implement its required orderings, the parallel system will not work correctly. ***MCM verification of designs and implementations is known for its difficulty, as MCMs are hard to specify unambiguously and their verification requires examination of a vast number of possible scenarios.*** MCM-related bugs in recent hardware and software have caused incorrect answers [26], unreliable software [8], and security exploits [17].

My PhD research (Figure 2) developed methodologies and tools for automated formal MCM verification at multiple levels of the hardware/software stack and at multiple points in the design timeline. A key idea in my work is the use of microarchitectural happens-before (μhb) graphs [24] to represent hardware event orderings and the use of SMT solvers to automatically examine all possible event orderings for a given scenario. I have developed tools for ***MCM verification of early-stage designs*** (PipeProof [2] and CCICheck [3]), ***detailed designs*** (RealityCheck [1]), and ***Verilog processor implementations*** (RTLCheck [4]). ***The combination of these tools enables progressive automated formal MCM verification of parallel processors.***

I also worked on TriCheck [6], a tool for MCM verification *across* layers that allows computer architects to check that their hardware designs will be compatible with the MCMs of high-level languages like C++. In addition, I worked on ILA-MCM, a framework [7] that combines instruction-level abstractions (ILAs) [31] for processing elements in heterogeneous SoCs with formal hardware ordering specifications [25] to create rich, detailed models of heterogenous SoCs. ***All my tools are open-sourced when published*** so that everyone may benefit from them. Below I expand on the major components of my dissertation work.

Verifying Early-Stage Designs: A hardware design must respect its ISA-level MCM across all possible programs (an infinite set) for correctness. However, no prior automated approach could conduct such *complete* MCM verification. To fill this gap, I developed ***PipeProof [2] to check that an axiomatic (i.e. constraint-based) microarchitectural model refines (i.e. correctly implements) its axiomatic ISA-level MCM specification for all programs.*** To cover the infinite search space required, I developed the novel ***Transitive Chain (TC) Abstraction*** to abstract the infinite set of programs into a finite set of cases. I also developed an algorithm based on CEGAR (counterexample-guided abstraction refinement) [12] to incrementally explore more cases as needed until the abstraction was strong enough to prove the hardware design correct (or find a bug). ***PipeProof was a best paper nominee at the MICRO-51 conference, and was recognized as an "Honorable Mention" for 2018 by the IEEE Micro Top Picks[1] selection committee.*** In addition, to the knowledge of my co-authors and myself, ***PipeProof is the first ever automated all-program refinement checking methodology and tool for pairs of axiomatic models of executions. Thus, PipeProof also enriches***

[1]IEEE Micro Top Picks uses peer review to select the 10-12 most influential computer architecture papers of the year.

the formal methods community by contributing new methods for reasoning about axiomatic models.

Shared-memory parallel systems use cache coherence protocols to ensure that all processor caches observe updates to memory locations. Coherence and MCMs are usually verified independently, preventing the detection of MCM bugs that arise due to their interaction. To fill this verification gap, I developed ***CCICheck*** [3] ***(best paper nominee at the MICRO-48 conference), the first methodology and tool capable of automatically detecting MCM bugs arising from the interaction of processor pipelines with cache coherence protocols. The development of CCICheck led to my discovery of a bug in TSO-CC*** [14, 13]***, a coherence protocol developed at the University of Edinburgh, and the subsequent fix of the bug by the protocol developers.*** A key idea in CCICheck was its novel ViCL (Value in Cache Lifetime) abstraction. ***ViCLs have been used in subsequent research to formally model cache behaviour relevant to side-channel attacks*** [33], enabling the automatic generation of security exploits (including new variants of Spectre and Meltdown).

Verifying Detailed Designs: As a design progresses, the early-stage monolithic specifications verified by PipeProof and CCICheck must evolve to include new design details. However, today's processors are complex heterogeneous parallel systems, where distinct teams (and possibly different vendors) each develop one or more components, and then connect them together to create the overall processor. Monolithic specifications of such processors would be extremely unwieldy. To fix this problem, ***I developed RealityCheck [1], which adds support for modularity and hierarchy to prior microarchitectural MCM verification work [25]. RealityCheck enables each team to specify the orderings of their component independently and then compose these specifications together to create a processor specification, mirroring real-world design.***

RealityCheck is also the first methodology and tool to enable scalable automated formal MCM verification of hardware designs. Verifying monolithic specifications does not scale because the entire complicated specification must be verified all at once. RealityCheck solves this problem through its support for modular abstraction. Each RealityCheck module can be verified against an interface specification of its external behaviour independently of the rest of the system. Then, when composing modules together to create the processor specification, each module can simply be represented by its interface. ***This use of modular abstraction decomposes overall processor MCM verification into multiple smaller verification problems, allowing MCM verification to scale even for large designs.***

Implementation Verification: Design-time verification ensures design correctness, but engineers also need to ensure that system implementations meet design requirements. For MCM verification of processor implementations, I developed ***RTLCheck*** [4] in collaboration with NVIDIA researchers. ***RTLCheck is the first automated tool for formal MCM verification of processor implementations written in Verilog RTL (register transfer language).*** Given a hardware design's ordering specification and a mapping from design components to their RTL equivalents, RTLCheck automatically translates the design's ordering properties to equivalent RTL properties for a given test program. The RTL properties thus generated can be checked by commercial RTL verifiers like Cadence's JasperGold, easing the integration of RTLCheck into industry workflows. RTLCheck uncovered a bug in the open-source RISC-V V-scale processor's memory implementation [26], which we reported to the processor developers. ***RTLCheck was recognized as an "Honorable Mention" for 2017 by the IEEE Micro Top Picks***[1] ***selection committee***.

Hardware-Software MCM Compatibility: My co-authors and I developed ***TriCheck*** [6], the first methodology and tool enabling engineers to automatically check their hardware designs for compatibility with programming language MCMs across a set of test programs. ***This work uncovered severe deficiencies in the MCM specification of the open-source RISC-V instruction set, which is used extensively by the computer hardware community.*** In response, the RISC-V Memory Model Working Group was formed, where I and other MCM experts worked together to develop a robust formal MCM specification for RISC-V. (A redesigned RISC-V MCM that fixed the issues we discovered was ratified in 2018.) ***TriCheck was recognized as an IEEE Micro Top Pick***[1] ***for the year 2017.***

TriCheck also discovered two counterexamples to a "proven-correct" compiler mapping [9] for C/C++ `atomic` variables to IBM Power and ARMv7 assembly language, where these "proven" mappings allowed incorrect hardware behaviour in violation of C and C++ requirements. Using TriCheck's counterexamples, I pinpointed the specific issue with the published proof that erroneously allowed the flawed mappings to be "proven" [5]. ***This discovery brought to light a critical issue with the C and C++ language MCMs*** that was later fixed [20]. This finding also led to ***my discovery of three MCM bugs (since fixed) in IBM's XL C++ compiler during a single-day in-person investigation at IBM***.

Future Research

Principled Design and Verification for a "Post-ISA" World: ***The end of Moore's Law and Dennard Scaling has caused a proliferation of heterogeneous parallel architectures***, as architects attempt to continue improving performance by creating specialised accelerators for certain types of computations [29]. Applications are increasingly breaking the hardware/software ISA abstraction to target accelerators directly, leading to a "post-ISA" world [27]. However, the interfaces between such emerging hardware and software still need to be explicitly and formally specified to make clear what the contract between hardware and software is. Furthermore, when accelerators are working in parallel with general-purpose cores to process data [29], a concurrency model for such hardware is also necessary. While progress has been made in this area [15, 18, 31], much work remains to be done to create robust abstractions and models for this new era. ***In the future, I will develop new hardware/software abstractions and concurrency models for emerging heterogeneous parallel architectures. These formal specifications will give software a clear and unambiguous model of hardware to target, while still allowing software to take advantage of hardware specialisation. They will ideally also provide some level of portability so that e.g. new iterations of an accelerator do not require modification of code or the software toolchain.*** My computer architecture knowledge and my MCM verification thesis work make me well-suited to develop such models.

Traditional verification flows that assume a single ISA for a processor also need to evolve for this new architectural landscape. My thesis work has already made advances in this regard: RealityCheck's modularity, hierarchy, and abstraction make it an excellent fit for heterogeneous parallel hardware, while the ILA-MCM framework I worked on is also designed to specify and verify heterogeneous parallel hardware. ***In the future, I will build on my thesis work to develop methodologies and tools for the formal verification of emerging heterogenous parallel systems. The hardware/software abstractions and concurrency models I will develop (see above) will serve as specifications for such verification.***

Automated Formal Specification Generation Through Higher-Level Hardware Design: ***Automated formal verification requires formal design specifications, which can be difficult to write for those without prior experience.*** For hardware, one alternative is to automatically extract these formal specifications from early-stage architectural prototypes. A problem with this notion is that most hardware description languages (HDLs) are very low-level (e.g. Verilog) and do not have higher-level semantic information.

Inspired by the Bluespec HDL and programming language type systems, ***I propose the use of higher-level HDLs with types that can encode semantic information. Automated tools can then scan a prototype's source code for elements of the relevant types and automatically generate a formal specification by analysing executions containing these elements.*** e.g. If generating MCM specifications, one might search for all structures of type `MCMElement` and then try to discover orderings between their events.

Progressive Verification of Other Domains: Progressive automated formal verification (Figure 1b) can be applied to other domains in need of stringent verification, like cloud-based distributed systems. These distributed systems are complex and highly concurrent. Their complexity results in bugs that cause node crashes and unavailable services [23], resulting in the loss of industry revenue. ***In the future, I will create methodologies and tools that enable progressive automated formal verification of distributed system designs and implementations. I am well-suited for such research because event orderings in distributed systems can be modelled using happens-before graphs similar to those in my dissertation work.***

Meanwhile, cyber-physical systems like self-driving cars and robots consist of computing devices tightly coupled with physical components that interact with the outside world. These systems are becoming more widespread and are incorporating Internet connectivity and machine learning, making them more prone to bugs. Such bugs can have disastrous effects, as in the case of the Boeing 737 Max disasters [16] and recent fatal accidents involving Tesla's Autopilot [21, 22]. ***In the future, I will develop approaches and tools for the progressive automated formal verification of cyber-physical systems to help prevent such bugs. I am particularly interested in verifying safety properties that deal with concurrency*** (which are similar to MCM properties from my thesis), as well as ***security properties*** (e.g. preventing remote hijacks of connected vehicles), which have been verified using techniques similar to those in my thesis [33, 7].

Overall, my research goal is to realise a world where formal methods are a standard part of every hardware and software engineer's verification toolkit. These formal verification tools will be capable of automatically proving an extensive set of properties while remaining easy to use. I believe that such mainstream use of formal methods will go a long way towards making hardware and software bugs a thing of the past.

References

[1] **Y. A. Manerkar**, D. Lustig, and M. Martonosi, "RealityCheck: Bringing modularity, hierarchy, and abstraction to automated microarchitectural memory consistency verification," in *Under Submission*, 2019.

[2] **Y. A. Manerkar**, D. Lustig, M. Martonosi, and A. Gupta, "PipeProof: Automated memory consistency proofs of microarchitectural specifications," in *51st International Symposium on Microarchitecture (MICRO)*, 2018.

[3] **Y. A. Manerkar**, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: Using μhb graphs to verify the coherence-consistency interface," in *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[4] ——, "RTLCheck: Verifying the memory consistency of RTL designs," in *50th International Symposium on Microarchitecture (MICRO)*, 2017.

[5] **Y. A. Manerkar**, C. Trippel, D. Lustig, M. Pellauer, and M. Martonosi, "Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings," *CoRR*, vol. abs/1611.01507, 2016. [Online]. Available: http://arxiv.org/abs/1611.01507

[6] C. Trippel, **Y. A. Manerkar**, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," in *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[7] H. Zhang, C. Trippel, **Y. A. Manerkar**, A. Gupta, M. Martonosi, and S. Malik, "ILA-MCM: integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification," in *Formal Methods in Computer Aided Design (FMCAD) 2018*, 2018. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8603015

[8] ARM, "Cortex-A9 MPCore, programmer advice notice, read-after-read hazards. ARM Reference 761319." 2011, http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf.

[9] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling C/C++ Concurrency: from C++11 to POWER," in *39th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.

[10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

[11] S. Bush, "RISC-V bugs found by princeton," 2017, https://www.electronicsweekly.com/open-source-engineering/risc-v-bugs-found-princeton-2017-04/.

[12] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *12th International Conference on Computer Aided Verification (CAV)*, 2000.

[13] M. Elver, "TSO-CC specification," 2015, http://homepages.inf.ed.ac.uk/s0787712/res/research/tsocc/tso-cc_spec.pdf.

[14] M. Elver and V. Nagarajan, "TSO-CC: consistency directed cache coherence for TSO," in *20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[15] H. Foundation, "HSA programmer reference manual specification 1.2," May 2018.

[16] D. Gelles and N. Kitroeff, "Boeing and F.A.A. faulted in damning report on 737 max certification," 2019, https://www.nytimes.com/2019/10/11/business/boeing-737-max.html.

[17] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 38–55.

[18] Khronos Group, "OpenCL 2.0." [Online]. Available: http://www.khronos.org/opencl

[19] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: http://arxiv.org/abs/1801.01203

[20] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in C/C++11," in *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[21] T. B. Lee, "There's growing evidence Tesla's autopilot handles lane dividers poorly," 2018, https://arstechnica.com/cars/2018/04/theres-growing-evidence-teslas-autopilot-handles-lane-dividers-poorly/.

[22] ——, "Autopilot was active when a Tesla crashed into a truck, killing driver," 2019, https://arstechnica.com/cars/2019/05/feds-autopilot-was-active-during-deadly-march-tesla-crash/.

[23] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "Dcatch: Automatically detecting distributed concurrency bugs in cloud systems," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 677–691. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037735

[24] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models," in *47th International Symposium on Microarchitecture (MICRO)*, 2014.

[25] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[26] **Y. A**. **Manerkar**, "Two stores in successive cycles cause the first one to be dropped," 2017, https://github.com/ucb-bar/vscale/issues/15.

[27] M. Martonosi, "New metrics and models for a post-ISA era: Managing complexity and scaling performance in heterogeneous parallelism and internet-of-things," *SIGMETRICS Perform. Eval. Rev.*, vol. 46, no. 1, pp. 20–20, Jun. 2018. [Online]. Available: http://doi.acm.org/10.1145/3292040.3219625

[28] Public Safety and Homeland Security Bureau, "Level 3 nationwide outage october 4, 2016," 2018, https://transition.fcc.gov/Daily_Releases/Daily_Business/2018/db0313/DOC-349661A1.pdf.

[29] V. J. Reddi and M. D. Hill, "Accelerator-level parallelism (alp)," 2019, https://www.sigarch.org/accelerator-level-parallelism/.

[30] D. Sorin, M. Hill, and D. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture, M. Hill, Ed. Morgan & Claypool Publishers, 2011.

[31] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, "Template-based synthesis of instruction-level abstractions for soc verification," in *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '15. Austin, TX: FMCAD Inc, 2015, pp. 160–167. [Online]. Available: http://dl.acm.org/citation.cfm?id=2893529.2893557

[32] The Coq development team, *The Coq proof assistant reference manual, version 8.0*, LogiCal Project, 2004. [Online]. Available: http://coq.inria.fr

[33] C. Trippel, D. Lustig, and M. Martonosi, "CheckMate: Automated synthesis of hardware exploits and security litmus tests," in *51st International Symposium on Microarchitecture (MICRO)*, 2018.