LSR

Yann Herklotz Grave

April 2022

Contents

1 Introduction

2 Background

Was there ever in anyone's life span a point free in time, devoid of memory, a night when choice was any more than the sum of all the choices gone before?

— Joan Didion, Run, River

aroistenaoirstenoiaresntoien

3 Formal Verification of High-Level Synthesis

Can you trust your high-level synthesis tool? As latency, throughput, and energy efficiency become increasingly important, custom hardware accelerators are being designed for numerous applications. Alas, designing these accelerators can be a tedious and error-prone process using a hardware description language (HDL) such as Verilog. An attractive alternative is *high-level synthesis* (HLS), in which hardware designs are automatically compiled from software written in a high-level language like C. Modern HLS tools such as LegUp [Can+11], Vivado HLS [Xil20], Intel i++ [Int20], and Bambu HLS [PF13] promise designs with comparable performance and energy-efficiency to those hand-written in an HDL [HG14; GW20; Pel+16], while offering the convenient abstractions and rich ecosystems of software development. But existing HLS tools cannot always guarantee that the hardware designs they produce are equivalent to the software they were given, and this undermines any reasoning conducted at the software level.

Indeed, there are reasons to doubt that HLS tools actually *do* always preserve equivalence. For instance, Vivado HLS has been shown to apply pipelining optimisations incorrectly[1] or to silently generate wrong code should the programmer stray outside the fragment of C that it supports.[2] Meanwhile, Lidbury et al. [Lid+15] had to abandon their attempt to fuzz-test Altera's (now Intel's) OpenCL compiler since it "either crashed or emitted an internal compiler error" on so many of their test inputs. More recently, Herklotz et al. [Her+21a] fuzz-tested three commercial HLS tools using Csmith [Yan+11], and despite restricting the generated programs to the C fragment explicitly supported by all the tools, they still found that on average 2.5% of test-cases were compiled to designs that behaved incorrectly.

[1] `https://bit.ly/vivado-hls-pipeline-bug`
[2] `https://bit.ly/vivado-hls-pointer-bug`

Existing workarounds Aware of the reliability shortcomings of HLS tools, hardware designers routinely check the generated hardware for functional correctness. This is commonly done by simulating the generated design against a large test-bench. But unless the test-bench covers all inputs exhaustively – which is often infeasible – there is a risk that bugs remain.

One alternative is to use *translation validation* [PSS98] to prove equivalence between the input program and the output design. Translation validation has been successfully applied to several HLS optimisations [YKM04; Kar+06; CK20; Ban+14; CKB19]. Nevertheless, it is an expensive task, especially for large designs, and it must be repeated every time the compiler is invoked. For example, the translation validation for Catapult C [Men20] may require several rounds of expert 'adjustments' [Cha20, p. 3] to the input C program before validation succeeds. And even when it succeeds, translation validation does not provide watertight guarantees unless the validator itself has been mechanically proven correct [e.g. TL08], which has not been the case in HLS tools to date.

Our position is that none of the above workarounds are necessary if the HLS tool can simply be trusted to work correctly.

Our solution We have designed a new HLS tool in the Coq theorem prover and proved that any output design it produces always has the same behaviour as its input program. Our tool, called Vericert, is automatically extracted to an OCaml program from Coq, which ensures that the object of the proof is the same as the implementation of the tool. Vericert is built by extending the CompCert verified C compiler [Ler09] with a new hardware-specific intermediate language and a Verilog back end. It supports most C constructs, including integer operations, function calls (which are all inlined), local arrays, structs, unions, and general control-flow statements, but currently excludes support for case statements, function pointers, recursive function calls, non-32-bit integers, floats, and global variables.

Contributions and Outline The contributions of this paper are as follows:

- We present Vericert, the first mechanically verified HLS tool that compiles C to Verilog. In Section **??**, we describe the design of Vericert, including certain optimisations related to memory accesses and division.
- We state the correctness theorem of Vericert with respect to an existing semantics

for Verilog due to Lööw and Myreen [LM19]. In Section **??**, we describe how we extended this semantics to make it suitable as an HLS target. We also describe how the Verilog semantics is integrated into CompCert's language execution model and its framework for performing simulation proofs. A mapping of CompCert's infinite memory model onto a finite Verilog array is also described.

- In Section **??**, we describe how we proved the correctness theorem. The proof follows standard CompCert techniques – forward simulations, intermediate specifications, and determinism results – but we encountered several challenges peculiar to our hardware-oriented setting. These include handling discrepancies between the byte-addressed memory assumed by the input software and the word-addressed memory that we implement in the output hardware, different handling of unsigned comparisons between C and Verilog, and carefully implementing memory reads and writes so that these behave properly as a RAM in hardware.

- In Section **??**, we evaluate Vericert on the PolyBench/C benchmark suite [Pou20], and compare the performance of our generated hardware against an existing, unverified HLS tool called LegUp [Can+11]. We show that Vericert generates hardware that is 27× slower (2× slower in the absence of division) and 1.1× larger than that generated by LegUp. This performance gap can be largely attributed to Vericert's current lack of support for instruction-level parallelism and the absence of an efficient, pipelined division operator. We intend to close this gap in the future by introducing (and verifying) HLS optimisations of our own, such as scheduling and memory analysis. This section also reports on our campaign to fuzz-test Vericert using over a hundred thousand random C programs generated by Csmith [Yan+11] in order to confirm that its correctness theorem is watertight.

Companion material Vericert is fully open source and available on GitHub at `https://github.com/ymherklotz/vericert`. A snapshot of the Vericert development is also available in a Zenodo repository [Her+21b].

4 WIP Static Scheduling

5 FW Loop Pipelining

6 FW Dynamic Scheduling

7 Schedule

8 Conclusion

Bibliography

[Ban+14] K. Banerjee et al. "Verification of Code Motion Techniques Using Value Propagation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.8 (Aug. 2014), pp. 1180–1193. ISSN: 1937-4151. DOI: 10.1109/TCAD.2014.2314392.

[Can+11] Andrew Canis et al. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems". In: *FPGA*. ACM, 2011, pp. 33–36. DOI: 10.1145/1950413.1950423.

[Cha20] Pankaj Chauhan. *Formally Ensuring Equivalence between C++ and RTL designs*. 2020. URL: https://bit.ly/2KbT0ki.

[CK20] R. Chouksey and C. Karfa. "Verification of Scheduling of Conditional Behaviors in High-Level Synthesis". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2020), pp. 1–14. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2020.2978242. URL: https://doi.org/10.1109/TVLSI.2020.2978242.

[CKB19] R. Chouksey, C. Karfa and P. Bhaduri. "Translation Validation of Code Motion Transformations Involving Loops". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.7 (July 2019), pp. 1378–1382. ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2846654.

[GW20] Stephane Gauthier and Zubair Wadood. *High-Level Synthesis: Can it outperform hand-coded HDL?* White paper. 2020. URL: https://info.silexica.com/high-level-synthesis/1.

[Her+21a] Yann Herklotz et al. "An Empirical Study of the Reliability of High-Level Synthesis Tools". In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021, pp. 219–223. DOI: 10.1109/FCCM51124.2021.00034.

[Her+21b] Yann Herklotz et al. *ymherklotz/vericert: Vericert v1.2.1*. Version v1.2.1. July 2021. DOI: 10.5281/zenodo.5093839. URL: https://doi.org/10.5281/zenodo.5093839.

[HG14] Ekawat Homsirikamol and Kris Gaj. "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study". In: *ReConFig*. IEEE, 2014, pp. 1–8. DOI: 10.1109/ReConFig.2014.7032504.

[Int20] Intel. *High-level Synthesis Compiler*. 2020. URL: https://intel.ly/2UDiWr5 (visited on 18/11/2020).

[Kar+06] C Karfa et al. "A Formal Verification Method of Scheduling in High-level Synthesis". In: *Proceedings of the 7th International Symposium on Quality Electronic Design*. ISQED '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 71–78. ISBN: 0-7695-2523-7. DOI: 10.1109/ISQED.2006.10.

[Ler09] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814.

[Lid+15] Christopher Lidbury et al. "Many-Core Compiler Fuzzing". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 65–76. ISBN: 9781450334686. DOI: 10.1145/2737924.2737986.

[LM19] Andreas Lööw and Magnus O. Myreen. "A Proof-producing Translator for Verilog Development in HOL". In: *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering*. FormaliSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 99–108. DOI: 10.1109/FormaliSE.2019.00020. URL: https://doi.org/10.1109/FormaliSE.2019.00020.

[Men20] Mentor. *Catapult High-Level Synthesis*. 2020. URL: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls (visited on 06/06/2020).

[Pel+16] Maxime Pelcat et al. "Design productivity of a high level synthesis compiler versus HDL". In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 2016, pp. 140–147. DOI: 10.1109/SAMOS.2016.7818341.

[PF13] Christian Pilato and Fabrizio Ferrandi. “Bambu: A modular framework for the high level synthesis of memory-intensive applications”. In: *FPL*. IEEE, 2013, pp. 1–4. DOI: 10.1109/FPL.2013.6645550.

[Pou20] Louis-Noël Pouchet. *PolyBench/C: the Polyhedral Benchmark suite*. 2020. URL: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[PSS98] A. Pnueli, M. Siegel and E. Singerman. “Translation validation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Berlin, Heidelberg: Springer, 1998, pp. 151–166. ISBN: 978-3-540-69753-4. DOI: 10.1007/BFb0054170.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. “Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: ACM, 2008, pp. 17–27. ISBN: 9781595936899. DOI: 10.1145/1328438.1328444.

[Xil20] Xilinx. *Vivado High-level Synthesis*. 2020. URL: https://bit.ly/39ereMx (visited on 20/07/2020).

[Yan+11] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 283–294. ISBN: 9781450306638. DOI: 10.1145/1993498.1993532. URL: https://doi.org/10.1145/1993498.1993532.

[YKM04] Youngsik Kim, S. Kopuri and N. Mansouri. “Automated formal verification of scheduling process using finite state machines with datapath (FSMD)”. In: *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720)*. Mar. 2004, pp. 110–115. DOI: 10.1109/ISQED.2004.1283659.