Resource Sharing for Verified High-Level Synthesis

Michalis Pardalos
Imperial College London, UK
Email: mpardalos@gmail.com

Yann Herklotz
Imperial College London, UK
Email: yann.herklotz15@imperial.ac.uk

John Wickerson
Imperial College London, UK
Email: j.wickerson@imperial.ac.uk

***Abstract*—High-level synthesis (HLS) is playing an ever-increasing role in hardware design, but concerns have been raised about its reliability. Seeking to address these, Herklotz et al. have developed an HLS compiler called Vericert that has been mechanically proven (using the Coq proof assistant) to output Verilog designs that are behaviourally equivalent to the input C program. Unfortunately, Vericert cannot compete performance-wise with established HLS tools, and a major reason for this is Vericert's complete lack of support for *resource sharing*.**

This paper introduces Vericert-Fun: Vericert extended with function-level resource sharing. Where original Vericert creates one block of hardware per function *call*, Vericert-Fun creates one block of hardware per function *definition*. To enable this, we extend Vericert's intermediate language HTL with the ability to represent *multiple* state machines, and we implement function calls by transferring control between these state machines. We are working to extend Vericert's correctness proof to cover the translation from C into this extended HTL and thence to Verilog. Benchmarking on the PolyBench/C suite indicates that Vericert-Fun generates hardware with about 0.8× the resource usage of Vericert's on average, with minimal impact on execution time.

I. Introduction

The drive for faster, more energy-efficient computation has led to a surge in demand for custom hardware accelerators. This, in turn, has led to interest in *high-level synthesis* (HLS) as a means to program these devices. Yet doubts have been raised about the reliability of the current crop of HLS tools. For instance, Herklotz et al. [12] found numerous miscompilation bugs in Xilinx Vivado HLS [24], the Intel HLS Compiler [15], and LegUp [5]. This unreliability can be a significant hindrance for developers, and it undermines the usefulness of HLS in safety- or security-critical settings.

Aiming to address this issue is Vericert [13], a new HLS tool whose correctness has been verified to the highest possible standard: a computer-checked proof that any Verilog design it produces will behave the same way as the C program given as input. Yet it is not enough for an HLS tool simply to be *correct*; the generated hardware must also enjoy high throughput, low latency, and good *area efficiency* – the last of which is the topic of this paper.

A common optimisation employed by HLS tools to improve area efficiency is *resource sharing*; that is, mapping multiple operations to the same hardware unit. Accordingly, our work adds function-level resource sharing to Vericert, yielding a new prototype HLS tool called 'Vericert-Fun'. In line with the aims of the Vericert project, work is ongoing to extend the correctness proof. The entire Vericert-Fun development is fully open-source [19], and more details about the implementation and proofs are available in a technical report [18].

978-1-6654-8332-2/22/$31.00 ©2022 IEEE

II. Background

a) The Coq proof assistant: Vericert is implemented in Coq [1], which means it consists of a collection of functions that define the compiler, together with the proof of a theorem that those definitions constitute a correct HLS tool. Coq mechanically checks this proof using a formal mathematical calculus, then translates the function definitions into OCaml code that can be compiled and executed. Developing software within a proof assistant like Coq is widely held to be the gold standard for correctness, and recent years have shown that substantial systems can be produced in this way, such as database systems [17], web servers [6], and OS kernels [11]. Coq has also been deployed in hardware design, both in academia [2, 3] and in industry [10]. It has even been applied specifically to HLS: Faissole et al. [8] used it to verify that HLS optimisations respect dependencies in the source code.

b) The CompCert verified C compiler: Among the most celebrated applications of Coq is CompCert [16], a lightly optimising C compiler with backend support for the Arm, x86, PowerPC, and Kalray VLIW architectures [23]. It transforms its input through a series of ten intermediate languages before generating the final output. The correctness proof of the entire compiler is formed by composing the correctness proofs of each of those passes. That the Csmith compiler testing tool [25] has found hundreds of bugs in GCC and LLVM but none in (the verified parts of) CompCert is a testament to the reliability of this development approach.

c) The Vericert verified HLS tool: Introduced by Herklotz et al. [13], Vericert is a verified C-to-Verilog HLS tool, built by extending CompCert with a new hardware-oriented intermediate language (called HTL) and a Verilog backend. Vericert branches off from CompCert at the intermediate language called *register-transfer language* (which we shall call '3AC', for 'three-address code', to avoid confusion with 'register-transfer level'). In 3AC, each function is represented as a numbered list of instructions with gotos – i.e., a control-flow graph (CFG). Vericert's compilation strategy is to treat this CFG as a state machine, with each instruction in the CFG being a state, and each edge in the CFG being a transition. The stack is implemented in a block of RAM, and program variables that do not have their address taken are mapped to hardware registers. More precisely, Vericert builds a *finite state*

machine with datapath (FSMD). This comprises two maps, both taking the current state as their input: the *control logic* map for determining the next state, and a *datapath* map for updating the RAM and registers. These state machines are captured in Vericert's new intermediate language, HTL. When Vericert compiles from HTL to the final Verilog output, these maps are converted from mathematical functions into syntactic Verilog case-statements, and placed inside always-blocks.

Vericert currently performs no significant optimisations beyond those inherited from CompCert's frontend. This results in performance generally about an order of magnitude slower than that achieved by comparable, unverified HLS tools. The overall Vericert flow is shown in Figure 1 (top). Note the 'inlining' step, which folds all function definitions into their call sites. This allows Vericert to make the simplifying assumption that there is only a single CFG, but has the unwanted effect of duplicating hardware. Vericert-Fun removes some of this inlining and hence some of the duplication.

d) Resource sharing in HLS: Resource sharing is a feature expected of most HLS compilers. In a typical HLS-generated architecture [7], several 'functional components' are selected from a library according to the needs of the specific design, and in the scheduling process, each operation is allotted a clock cycle in which the required components are all available. Given the need to mechanically verify the correctness of our implementation, Vericert-Fun follows a simpler approach: we share resources at the granularity of entire functions, rather than individual operations. Function-level resource sharing is implemented in commercial HLS compilers such as the Intel HLS compiler [15] or Xilinx Vitis [24], and is guided by the programmer through pragmas.

Perna et al. [20] developed a verified HLS tool for the Handel-C language, but, like Vericert, they did not implement function-level resource sharing, instead arranging that "all procedure and function calls are expanded in the front-end".

III. Implementation of Vericert-Fun

We now explain the implementation of Vericert-Fun using Figure 2 as a worked example. The overall flow is shown in Figure 1 (bottom): we avoid inlining the function calls at the 3AC level (except in certain circumstances described below), instead maintaining one state machine per function. All the state machines run simultaneously, and function calls are implemented by sending messages between them. We then combine all of these state machines into a single Verilog module after renaming variables to avoid clashes.

Figure 3 shows the 3AC representation of that C program, as obtained by the CompCert frontend. We see two CFGs, one per function. The control flow in this example is straightforward, but in general, 3AC programs can contain unrestricted gotos. The nodes of the CFGs are numbered in reverse, as are the parameters of the `add` function, following CompCert convention. Figure 4 depicts the result of converting those CFGs into two state machines. The conversion of 3AC instructions into Verilog instructions has been described already by Herklotz et al. [13]; what is new here is the handling of function

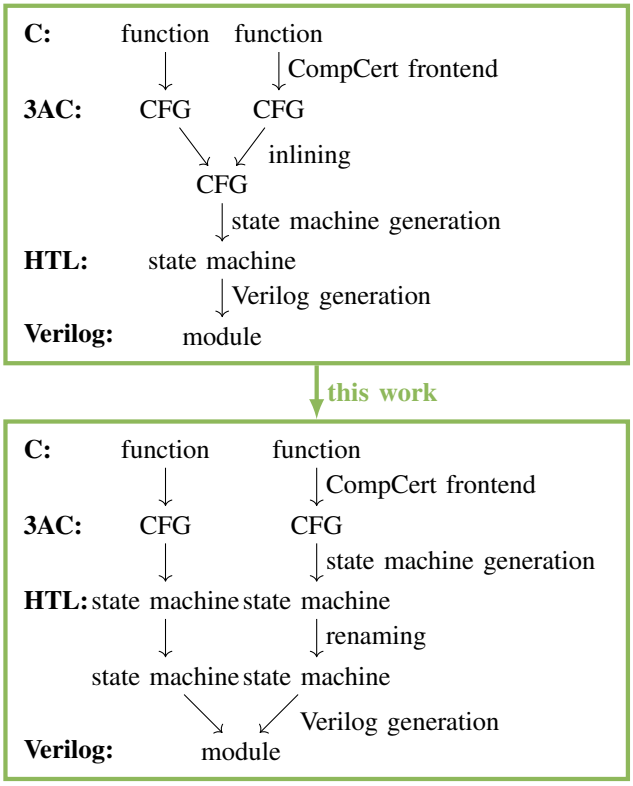

Fig. 1: Key compilation passes and intermediate languages in Vericert [13] (top) and in Vericert-Fun (bottom)

```
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int v = 0;
  v = add(v, 1);
  v = add(v, 2);
  return v;
}
```

Fig. 2: A simple C program with each instruction colour-coded so it can be tracked through the compilation stages

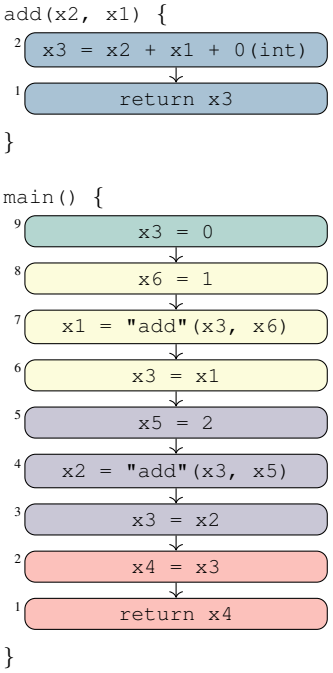

Fig. 3: 3AC representation comprising two CFGs

calls, so the following concentrates on that aspect. Note that "$*\rightarrow\langle node\rangle$" stands for edges from all nodes to $\langle node\rangle$. The solid edges within the two state machines indicate the transfer of control between states, while the dashed edges between the state machines are more 'fictional'. The ground truth is that both state machines run continuously, but it is convenient to think that only one machine does useful work at a time. So, the dashed edges indicate when the 'active' machine changes.

In more detail: Execution begins in state 9 of the `main` machine, and proceeds through successive states until it reaches state 7, in which the `add` machine's `rst` signal is set. This causes the `add` machine to advance to state 2. When `main` advances to state 12, that `rst` signal is unset; `add` then begins its computation while `main` spins in state 12. When `add` has finished (state 1), it sets its `fin` signal, which allows `main` to leave state 12. Finally, `add` unsets its `fin` and waits in state 3 for the next call. The same event sequence can also be understood using the timing diagram in Figure 5, in which red lines indicate unspecified values. We see that calls are initiated by triggering the `rst` signal of the called module and that a function returns by setting its own `fin` register.

One technical challenge we encountered in the implementation of Vericert-Fun has to do with the fact that the called and callee state machines need to modify each other's variables. This is problematic because each function is translated independently, and hence the register names used in the other state machines may not be available. We work around this by introducing an additional component to our state machines: an '`extern_ctrl`' mapping from local register names to pairs of module identifiers and roles in those modules. For instance, the first entry in `extern_ctrl` in Figure 4 tells us that the `add_0_rst` register used by `main` should resolve to whichever register plays the role of 'reset' in `add`. Once all the state machines have been generated, we erase `extern_ctrl`. We do this in two steps. First, we rename all registers to be globally unique, which avoids unintended conflicts between registers in different modules (register names can only be assumed unique within their own module). We then rename all registers mentioned in `extern_ctrl` to the name of the actual register they target.

A second technical challenge we encountered in the implementation of Vericert-Fun has to do with an assumption made in Vericert's correctness proof: that all pointers are stored as offsets from the main function's stack frame. This assumption was reasonable for Vericert because after full inlining, the main function is the only function. This assumption is baked into several of Vericert's most complicated lemmas, including the correctness proof for load and store instructions, and so we have not sought to lift it in our current prototype of Vericert-Fun. Instead, we have made the compromise of only *partially* eliminating the inlining pass. That is: Vericert-Fun inlines all functions that contain `load`, `store` or `call` instructions. Thus, the benefits of resource sharing are currently only enjoyed by functions that do not contain load or store instructions and do not call other functions.

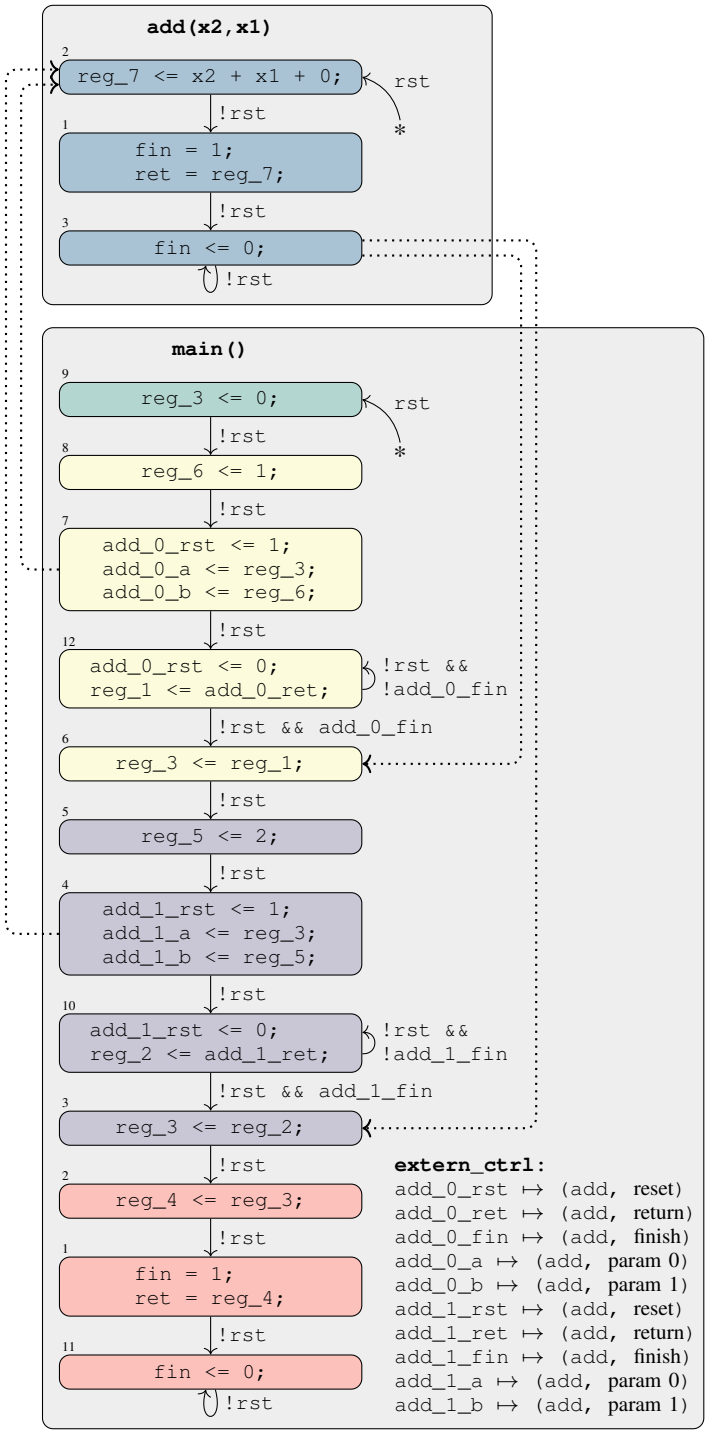

Fig. 4: HTL representation comprising two state machines

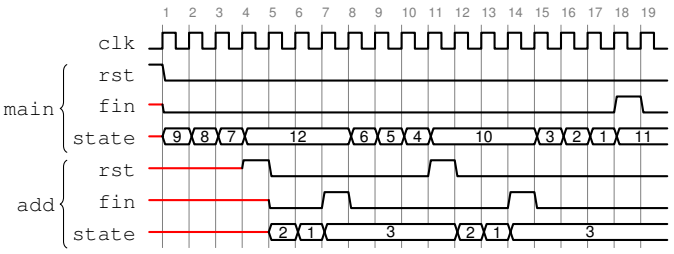

Fig. 5: Timing diagram for the state machines in Figure 4

IV. Proving Vericert-Fun correct

The CompCert correctness theorem [16] states that every behaviour of the compiled program is also a behaviour of the source program. Herklotz et al. [13] adapted this theorem for HLS by replacing 'compiled program' with 'generated Verilog design'. In both cases, a formal semantics is required for the source and target languages. Vericert-Fun targets the same fragment of the Verilog language as Herklotz et al. already mechanised in Coq, so no changes are required there.

Where changes *are* required is in the semantics of the intermediate language HTL, which sits between CompCert's 3AC and the final Verilog. When Herklotz et al. designed HTL, they did not include a semantics for function calls because they assumed all function calls would already have been inlined. We have extended HTL so that its semantics is additionally parameterised by an environment that maps function names to state machines. Our semantics for function calls looks up the named function in this environment, activates the corresponding state machine, and pushes a new stack frame, and our semantics for return statements pops the current stack frame and reactivates the caller's state machine.

At the point of writing, the correctness of Vericert-Fun from C to HTL has been mostly proven: basic instructions and function calls are proven correct, but proofs of load and store instructions still lack some key invariants. The pass that renames variables in HTL is yet to be proven, as is the Verilog-generation pass. To give a rough idea of the scale and complexity of our work: the implementation of Vericert-Fun involved adding or changing about 700 lines of Coq code in Vericert and took the first author 4 months. The correctness proof has so far required about 2300 lines of additions or changes to the Coq code and 8 person-months of work.

V. Performance evaluation

We now compare the quality of the hardware generated by Vericert-Fun against that of Vericert. We use the open-source (but unverified) Bambu tool [9, 21] as a baseline. We run Bambu (version 0.9.6) in the `BAMBU_AREA` configuration, which optimises for area ahead of latency, but do not provide any additional pragmas to control the HLS process. Following Herklotz et al. [13], we use the PolyBench/C benchmark suite [22] with division and modulo replaced with iterative software implementations because Vericert does not handle them efficiently. That is, `a/b` and `c%d` are textually replaced with `div(a,b)` and `mod(c,d)`. These `div` and `mod` functions are the only function calls that are not inlined. We used the Icarus Verilog simulator to determine the cycle counts of the generated designs. We used Xilinx Vivado 2017.1, targeting a Xilinx 7-series FPGA (XC7K70T) to determine area usage (measured in slices) and fmax.

Figure 6 compares the hardware generated by Vericert-Fun with that of Vericert, using Bambu as a baseline. The top graph compares the area usage. We observe a substantial reduction in area usage across the benchmark programs, with Vericert consistently using more area than Bambu (1.5× on average) and Vericert-Fun requiring less area than Vericert (0.8× on average), but still more than Bambu (1.2× on average). As expected, the benchmarks with several function calls (mvt, 2mm, 3mm, ludcmp) enjoy the biggest area savings, while those with only one function call (heat-3d, nussinov) require slightly more area because of the extra circuitry required. The bottom graph compares the execution time. We observe that Vericert-Fun generates hardware that is slightly (about 4%) slower than Vericert's, which can be attributed to the latency overhead of performing a function call. Hardware from Vericert and Vericert-Fun is significantly slower than Bambu's, which can be attributed to Vericert employing far fewer optimisations than the unverified Bambu tool.

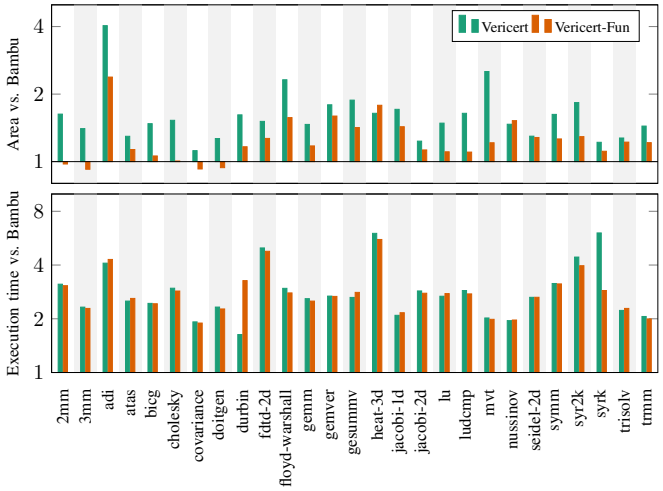

Fig. 6: Vericert-Fun vs Vericert-Fun, with Bambu as a baseline.

VI. Future work

Our immediate priority is to complete Vericert-Fun's correctness proof. In the medium term, we intend to improve our implementation of resource sharing by dropping the requirement to inline functions that access pointers or perform function calls; we anticipate that this will lead to further area savings and also allow Vericert-Fun to be evaluated on benchmarks with more interesting call graphs. We would also like Vericert-Fun to generate designs with one Verilog module per C function, as this is more idiomatic than cramming all the state machines into a single module; we did not do this yet because it requires extending the Verilog semantics to handle multiple modules. It would also be interesting to implement *selective* inlining of functions [14], either guided by heuristics or by programmer-supplied pragmas. It is worth noting that having proven inlining correct in general, the *amount* of inlining can be adjusted without affecting the correctness proof. Longer term, we would like to combine resource sharing with operation scheduling, i.e. resource-constrained scheduling [4].

Acknowledgments

This work was financially supported by the EPSRC via the Research Institute for Verified Trustworthy Software Systems (VeTSS) and the IRIS Programme Grant (EP/R006865/1).

References

[1] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[2] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. ACM, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[3] Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 213–228. https://doi.org/10.1007/978-3-642-39799-8_14

[4] Andrew Canis, Stephen Dean Brown, and Jason Helge Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*. IEEE, 1–8. https://doi.org/10.1109/FPL.2014.6927490

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[6] Adam Chlipala. 2015. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 609–622. https://doi.org/10.1145/2676726.2677003

[7] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andrés Takach. 2009. An Introduction to High-Level Synthesis. *IEEE Des. Test Comput.* 26, 4 (2009), 8–17. https://doi.org/10.1109/MDT.2009.69

[8] Florian Faissole, George A. Constantinides, and David Thomas. 2019. Formalizing Loop-Carried Dependencies in Coq for High-Level Synthesis. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 315–315. https://doi.org/10.1109/FCCM.2019.00056

[9] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1327–1330. https://doi.org/10.1109/DAC18074.2021.9586110

[10] Google. 2021. Project Silver Oak: Formal specification and verification of hardware, especially for security and privacy. https://github.com/project-oak/silveroak.

[11] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 653–669. https://doi.org/10.5555/3026877.3026928

[12] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. 2021. 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)*. 219–223. https://doi.org/10.1109/FCCM51124.2021.00034

[13] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485494

[14] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Dean Brown, and Jason Helge Anderson. 2015. The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware. *ACM Trans. Reconfigurable Technol. Syst.* 8, 3 (2015), 14:1–14:26. https://doi.org/10.1145/2629547

[15] Intel. 2022. Intel High Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[16] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[17] J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 237–248. https://doi.org/10.1145/1706299.1706329

[18] Michalis Pardalos. 2021. *Formally verified resource sharing for High Level Synthesis*. Master's thesis. https://mpardalos.xyz/thesis.pdf

[19] Michalis Pardalos, Yann Herklotz, and John Wickerson. 2022. Public repository for Vericert-Fun. https://github.com/mpardalos/Vericert-Fun or https://doi.org/10.5281/zenodo.5866708.

[20] Juan Perna and Jim Woodcock. 2012. Mechanised

Wire-Wise Verification of Handel-C Synthesis. *Science of Computer Programming* 77, 4 (2012), 424 – 443. https://doi.org/10.1016/j.scico.2010.02.007

[21] C. Pilato and F. Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–4. https://doi.org/10.1109/FPL.2013.6645550

[22] Louis-Noël Pouchet. 2016. PolyBench/C: the Polyhedral Benchmark suite. https://sourceforge.net/projects/polybench/

[23] Cyril Six, Sylvain Boulmé, and David Monniaux. 2020. Certified and efficient instruction scheduling: application to interlocked VLIW processors. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 129:1–129:29. https://doi.org/10.1145/3428197

[24] Xilinx Inc. 2022. Vitis HLS. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532