Synthesizing Legacy String Code for FPGAs Using Bounded Automata Learning

Kevin Angstadt
St. Lawrence University

Tommy Tracy II
University of Virginia

Kevin Skadron
University of Virginia

Jean-Baptiste Jeannin
University of Michigan

Westley Weimer
University of Michigan

***Abstract*—The adoption of hardware accelerators, such as FPGAs, into general-purpose computation pipelines continues to expand, but programming models for these devices lag far behind their CPU counterparts. While high-level synthesis (HLS) can help port some legacy software, many programs perform poorly without manual, architecture-specific optimization. We propose an end-to-end approach combining dynamic and static analyses to learn a model of functional behavior for off-the-shelf legacy code and synthesize a hardware description from this model. Our prototype implementation can correctly learn functionality for string kernels that recognize regular languages and provides a near approximation otherwise. We evaluate our prototype tool on a benchmark suite of real-world, legacy string functions mined from GitHub and successfully synthesize—without modification or annotation—over 80% (72% exactly and a further 11% approximately). Traditional HLS, only after extensive modification and custom testbench generation, can synthesize the same number of benchmarks, but with results that have higher hardware requirements and lower maximum clock rates.**

Introduction

The confluence of several factors, including the significant increase in data collection, demands for real-time analyses by business leaders, the effective end of Dennard Scaling, and the slowdown in Moore's Law density scaling, have led to the increased use of hardware accelerators, including Field-Programmable Gate Arrays (FPGAs), which offer a reconfigurable substrate of logic blocks and storage elements. FPGAs in high-performance computing are primarily used to synthesize application- or domain-specific accelerators; however, successful adoption currently requires additional architectural knowledge for effective programming and configuration. Frameworks that reduce the need for architectural knowledge or manual optimization would boost the adoption of these devices.

Adopting hardware accelerators into existing application workflows requires porting code to these new programming models. Unfortunately, porting legacy code remains difficult. The primary programming model for FPGAs remains Hardware Description Languages (HDLs) such as Verilog and VHDL. HDLs have a level of abstraction akin to assembly-level development on traditional CPU architectures. Despite providing high throughputs, programming with HDLs can be tedious to write correctly and optimize.

Higher levels of abstraction for programming FPGAs have been achieved with high-level synthesis (HLS) [1], languages such as OpenCL,[1] and frameworks such as Xilinx's SDAccel.[2] While HLS may allow existing code to compile for FPGAs, it still requires low-level knowledge of the underlying architecture to allow for efficient implementation and execution of applications [2]. Therefore, there is a need for a new programming model that supports porting of legacy code, admits performant execution on hardware accelerators, and does not rely on developer architectural knowledge.

In this article, we extend our AUTOMATASYNTH[3] framework to support end-to-end (source-to-FPGA) synthesis of *Boolean string kernels* (functions from strings to booleans) and present a new empirical comparison with HLS. These kernels represent a class of functions with applications ranging from high-energy physics to network security and bioinformatics. AUTOMATASYNTH uses a new approach for synthesizing and executing code on FPGAs. Unlike HLS, which statically analyzes a program to produce a hardware design, AUTOMATASYNTH both dynamically observes and statically analyzes program behavior to synthesize a *functionally-equivalent* hardware design. Our approach combines recent advances in state machine acceleration [3], query-based learning of state machines [4], and formal methods [5]. Our algorithm features a formal correctness proof [6].

We evaluate our end-to-end prototype on a suite of string kernels mined from GitHub. We compare the performance of AUTOMATASYNTH with a commercial HLS tool. Our evaluation demonstrates that AUTOMATASYNTH outperforms current-generation HLS tools in terms of hardware utilization and especially in programmer time and effort.

[1] https://www.khronos.org/opencl/

[2] https://www.xilinx.com/products/design-tools/legacy-tools/sdaccel.html

[3] https://github.com/kevinaangstadt/automata-synth

BACKGROUND AND RELATED WORK

AUTOMATASYNTH employs a novel combination of tools and techniques from multiple computing disciplines. To the best of our knowledge, this is the first framework to combine model learning algorithms, software verification, and architectures for high-performance automata processing. We position our work in the context of related efforts from each area.

Finite Automata

We model the behavior of legacy source code using *deterministic finite automata* (DFAs) to enable efficient acceleration with FPGAs. A DFA processes input data by repeated application of a state transition function with each subsequent symbol in the input string. If an accepting state is active after all input characters have been processed, the DFA *accepts* the input (i.e., the input matches the pattern encoded by the DFA).

Accelerators for Finite Automata

There is substantial research on the acceleration of finite automata computation. Reconfigurable computing has emerged as an effective platform for accelerating this form of computation, and automata have enabled the acceleration of a wide variety of applications across many domains [3].

Rahimi et al.'s Grapefruit framework enables high-throughput automata processing using modern FPGAs [3]. While supporting high performance, input problems must be phrased in an explicit state machine model, which is uncommon in extant software. Indeed, writing an automaton has been demonstrated to be error-prone and difficult [7], thus leaving an abstraction gap and hindering widespread adoption of automata processing accelerators.

State Machine Learning Algorithms

State machine learning attempts to learn a state machine representation of a software or

hardware system. These algorithms are a subset of *model learning* and have been the subject of study for several decades [8]. The most common approach is to use *active learning* in which the model is learned by performing experiments (tests) on the software or system to be learned. State machine learning has been applied widely, from internet banking to describing machine learning classifiers [8]. Learning an equivalent state machine from software remains challenging, and most approaches employ approximations [4].

Program Synthesis and Verification

Program synthesis is a holistic term for automatically generating software from some input description. Many approaches employ *counterexample-guided inductive synthesis* (CEGIS) to produce a solution. CEGIS iteratively constructs candidate solutions that are tested (typically via formal methods) for equivalence. We note that CEGIS is largely equivalent to the techniques used in the model learning community.

There has been significant research and engineering effort applied to making these techniques scalable, including using bounded or iterative techniques to address recursive control flow [5]. Most closely related to our work has been the use of bounded model checking to verify string-processing web applications; however, this work often focused on secure information flow rather than constraints over strings [9].

A related body of research focuses on extracting program behavior from legacy code for acceleration using domain-specific languages (DSLs), an approach referred to as *verified lifting*. General lifting approaches, however, often target CPUs and GPUs rather than FPGAs [10].

High-Level Synthesis for FPGAs

High-Level Synthesis (HLS) allows for FPGA development at a much higher level of abstraction (e.g., C) than HDLs [1] and has been demonstrated to reduce development time [11]. However, the performance of designs constructed using HLS can be unimpressive, requiring significant annotation and optimization [2]. HLS tools may also not support all features of the source language (e.g., recursion and dynamic data structures), meaning that legacy code must be refactored before HLS can apply. Although HLS does reduce the effort required to target FPGAs over HDLs, there is still a significant amount of work required to adapt C-like code to run on FPGAs. Unlike HLS, our approach of decoupling existing software design with the final FPGA representation allows AUTOMATASYNTH to generate peformant code with minimal rework.

LEARNING STATE MACHINES FROM LEGACY CODE

We present AUTOMATASYNTH, a framework for synthesizing hardware descriptions from off-the-shelf, legacy code implementing regular languages. Our approach extends Angluin's L* algorithm [4] by (1) using bounded software model checking with incremental unrolling to implement one of its assumptions, (2) using software testing to implement another of its assumptions, and (3) transforming learned models into DFAs for hardware synthesis.

L* Primer

Because many of our framework decisions and results depend on the formulation of Dana Angluin's foundational L* algorithm [4], we sketch it briefly.

At its core, the L* algorithm relies on a *minimally adequate teacher* (MAT) to answer two kinds of queries about a held-out regular language, L. First, the MAT must answer *membership* queries, yielding a Boolean value indicating if the queried string is a member of L. Second, the MAT must answer *conjecture* or *termination* queries.[4] Given a candidate regular language A, the MAT responds with `true` if $A = L$ or responds with a *counterexample* string for which A and L differ. (Since automata learning is used in applications where L is not a DFA, this query typically cannot be resolved by standard DFA equivalence checking.)

Member queries are used to construct a structured *observation table*. Then, the table is directly transformed into a candidate automaton for a termination query. If the MAT responds with a counterexample, the counterexample string and its prefixes are added to the observation table. The

[4]These are also called equivalence queries, but we avoid this term to prevent confusion with similar uses of the term in software verification.

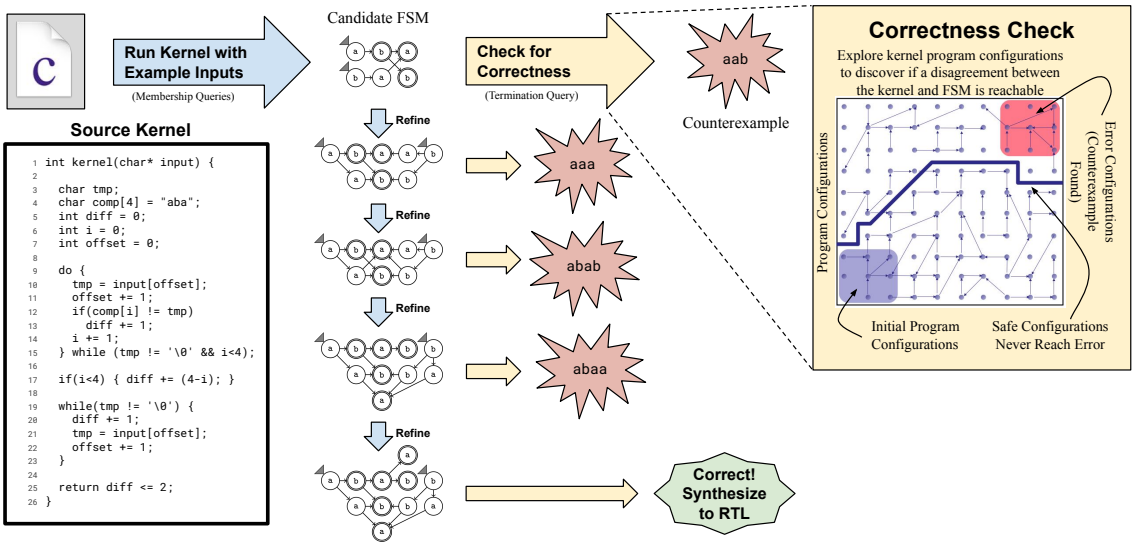

Figure 1. Example execution of AUTOMATASYNTH. The kernel function returns `true` for all strings within a hamming distance 2 of "aba". An initial automaton (triangles indicate starting states) is learned by running the kernel with example inputs. It is then checked for correctness using software verification to discover inputs that disagree. Four further refinements introduce new states and transitions through automated membership and termination queries. In this example, all counterexamples are strings the automaton should accept, but does not (false negatives). The final automaton passes all checks and is ready for synthesis to RTL.

process repeats until the MAT responds to a termination query affirmatively. The final automaton is minimal and accepts the learned language.

AUTOMATASYNTH Problem Description

In this subsection, we formalize the problem of learning a state machine from a legacy *Boolean string kernel*: a function that takes one string argument and returns a Boolean value:

$$\texttt{kernel} : \texttt{string} \rightarrow \texttt{bool}$$

We assume that the source code for this function is provided and that the function halts and returns a value on all inputs (i.e., `kernel` is an algorithm). If `kernel` recognizes a regular language, AUTOMATASYNTH returns a state machine, M, with equivalent behavior to `kernel`: for all $s \in \Sigma^*$, $M(s) = \texttt{kernel}(s)$. For runs which exceed a resource budget or expose incompleteness in the underlying theorem prover (including functions that are non-regular), our prototype implementation alerts and provides *approximate* equivalence, where $M(s) = \texttt{kernel}(s)$ when the length of s is less than an arbitrary fixed length. We have formally proven that our framework produces an equivalent DFA for input kernels that recognize regular languages [6]. Our empirical evaluation demonstrates that real-world legacy string kernels either recognize regular languages, or our tool can produce approximations of the original functions.

Figure 1 presents a worked example of AUTOMATASYNTH learning a kernel function that matches all strings within a hamming distance of two from "aba". Execution of the kernel (membership queries) produces an initial candidate automaton, which is correct for a subset of inputs. In this example, five termination queries (and additional membership queries) iteratively refine the candidate automaton.

Using Source Code as a MAT

We extend Angluin's L* algorithm to learn a DFA representation of a legacy string kernel by constructing a MAT that can answer membership and termination queries for a string kernel.

Membership Queries.
We observe that a membership query for a string, s, may be implemented by executing the legacy kernel on s: the result returned by the function is the answer to the query. For C-style languages,

we interpret Boolean values in the standard way (i.e., 0 is `false` and other values are `true`). We found that compiling the kernel to a shared object and then invoking the function dynamically provided the best stability in our experiments.

Termination Queries.

At the heart of our problem formulation is the challenge that a legacy string kernel does not admit a direct means for answering termination queries. For example, our initial efforts found testing alone (which is applicable for some domains of model learning [8]) often incorrectly answered termination queries in this use case. Our insight is that verification strategies from software model checking can test for equivalence between the kernel and a candidate automaton.

Traditionally in verification, equivalence would be proven using bisimulation or interleaving of the automaton and the source kernel. However, this formulation presupposes that the "state transitions" are directly encoded in the source code and can be aligned with the state transitions in the candidate automaton. We do not make this assumption in our problem definition, and we prefer an approach that does not require manual annotation. To broadly support legacy code, we do not even assume that the states of the equivalent automaton are visited "in order" during the execution of the legacy kernel.

We observe that a counterexample $t \in \Sigma^+$ is in either in $L(\texttt{kernel})$ or $L(M)$ but not in both, and thus will always satisfy the constraint $t \in L(\texttt{kernel}) \oplus L(M)$, where $\oplus$ is the symmetric difference operator. Therefore, we ask the software verifier to prove that there is no execution of `kernel` such that `kernel` and the candidate machine *disagree* on an answer. This formulation allows verification *without* directly encoding bisimulation. To test this reachability property, we use a novel combination of bounded model checking with incremental loop unrolling augmented with a string constraint solver. Our prototype uses the CPAChecker verifier [12], which we extended to support the draft SMT-LIB strings theory interface.[5]

[5]http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml

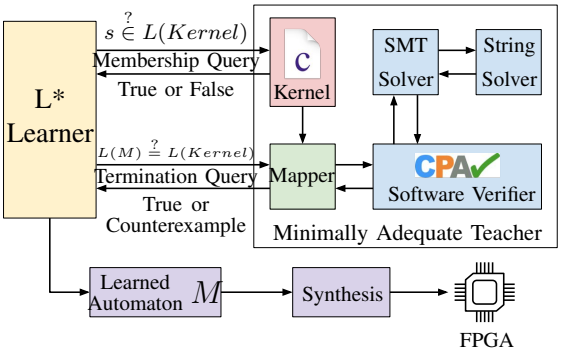

Figure 2. AUTOMATASYNTH System Architecture. The MAT uses the legacy kernel to answer membership queries. The mapper combines the kernel and candidate automaton to produce a software verification problem. Using bounded software model checking combined with string decision procedures, we search for a counterexample that distinguishes the target language from the language of the candidate automaton. Finally, we synthesize the learned automaton for execution on an FPGA.

Synthesizing Hardware Descriptions from Automata

Once a state machine has been learned using the L* algorithm with our custom MAT, the kernel is now amenable to acceleration. We convert the learned automaton to a hardware description and synthesize the design for loading onto an FPGA using Grapefruit [3]. Grapefruit does not attempt to replace traditional HLS and its associated optimizations for the general case. Instead, it replaces HLS with an optimized RTL mapping of the learned automaton. HLS optimizations such as memory partitioning and loop transformations don't have direct correspondences in Grapefruit and AUTOMATASYNTH because there is no one-to-one mapping from source to RTL.

System Architecture

Figure 2 depicts the high-level system architecture of our framework. The L* learner (left) queries a MAT (right) consisting of the legacy source code, software model checker, SMT solver, and string decision procedure. The legacy string kernel is used by the MAT to answer membership queries. Termination queries are transformed by a mapper into a software verification problem that searches for a string

that distinguishes the language of a candidate automaton from the target language implemented in the kernel. The output of the Learner is a DFA that encodes the same computation as the Kernel. We use this DFA to synthesize a hardware design for execution on an FPGA.

EXPERIMENTAL METHODOLOGY

In this section, we describe our process for selecting real-world, legacy string kernel benchmarks as well as our experimental setup for the evaluation described in the following section.

Benchmark Selection

In our evaluation, we compare the performance of AUTOMATASYNTH with that of a commercial HLS tool. We construct our benchmark suite by mining legacy string kernels from the most popular open-source software projects on GitHub. In total, we considered 26 repositories and mined 973 separate string kernel functions. After filtering for duplicates and a manual analysis to identify functions that return Boolean values, we collected 18 meaningfully-distinct real-world benchmarks.

The first three columns of Table 1 provide an overview of these string kernels. We use the function name to refer to each benchmark and also indicate the source project for each. Lines of code (LOC) provides a count of the total number of non-comment lines in the post-processed version of the benchmark.

High-Level Synthesis Workflow

We evaluate the performance of our synthesized automata against those generated by Xilinx Vitis HLS version 2020.1. We run HLS on the source code targeting the Alveo U280 accelerator card and using the Vivado IP Flow Target. We also created a testbench for each source file for verification. HLS generates Verilog and data files. To gather utilization and timing results we perform synthesis, placement and routing (in out-of-context mode).

We applied a standardized series of transformations to each benchmark kernel to support HLS synthesis and verification. First, if the kernel operates over a fixed set of indices, we bounded the input length in HLS. Next, if the kernel used standard string library code (e.g., `strcmp`), we added a synthesizable version of the source. Finally, if the kernel operated over an unbounded array, we ported the kernel to C++ to allow use of an HLS *stream* abstraction. In practice, we found that this transformation is nontrivial and required significant programmer effort and expertise. For consistency in timing with our execution of AUTOMATASYNTH, we limited ourselves to one day (approximately six hours of dedicated work time) per kernel to port, test, and optimize using HLS.

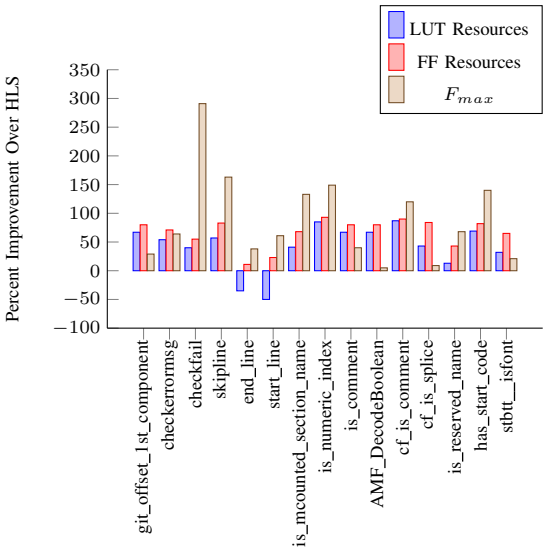

Figure 3. Differences in resource and time utilization. Higher bars show AUTOMATASYNTH using *fewer* resources and running at a higher frequency than HLS.

Experimental Setup

All executions of AUTOMATASYNTH use an Ubuntu 16.04 Linux server with a 3.0 GHz Intel Xeon E5-2623-v3 with four physical cores and 16 GB of RAM and a maximum time budget of 24 hours. We target the Xilinx Alveo U280 FPGA for both automata generated by AUTOMATASYNTH and also Verilog generated by Vitis HLS. The U280 is a data center accelerator, one of the highest performance cards offered by Xilinx.

EVALUATION

In this section, we first evaluate the correctness of the state machines generated by AUTOMATASYNTH and report runtime and query counts. Second, we present a new comparison of the performance of AUTOMATASYNTH and Xilinx High-Level Synthesis (HLS).

Table 1. Experimental Results of AUTOMATASYNTH on Benchmark Suite of Real-World, Legacy String Kernels

Benchmark	Project	LOC	Membership Queries	Term. Queries	Number of States	Total Runtime (m)	Correct	Approx. HLS Dev. Time (m)
`git_offset_1st_component`	*Git*: Revision control system	6	4,090	2	2	<1	✓	60
`is_encoding_utf8`		38	–	–	–	–	✘†	✘
`checkerrormsg`	*jq*: Command-line JSON processor	4	32,664	2	15	1,437	✓*	60
`checkfail`		14	189,013	3	35	1,438	✓*	60
`skipline`		17	7,663	3	3	5	✓	60
`end_line`	*Linux*: OS kernel	11	510,623	4	44	492	✓	60
`start_line`		11	206,613	2	46	80	Approx.	60
`is_mcounted_section_name`		54	672,041	7	57	1438	Approx.	120
`is_numeric_index`	*MASSCAN*: IP port scanner	17	10,727	3	4	5	✓	120
`is_comment`		11	4,090	2	2	<1	✓	60
`AMF_DecodeBoolean`	*OBS Studio*: Live streaming and recording software	2	2,557	2	2	<1	✓	60
`cf_is_comment`		28	4,599	2	4	5	✓	240
`cf_is_splice`		22	1,913	2	4	<1	✓	60
`is_reserved_name`		39	350,705	8	42	1,424	✓	120
`has_start_code`		18	10,213	2	7	<1	✓	60
`number_is_valid`	*openpilot*: Open-source driving agent	72	–	–	–	–	✘‡	✘
`utf8_validate`		72	–	–	–	–	✘§	✘
`stbtt__isfont`		24	79,598	5	19	<1	✓	60

*AUTOMATASYNTH warned of a potential approximate solution due to timeout, but manual analysis confirmed correctness
†Requires `strcasecmp` support ‡Requires `strtod` support §Performs math on characters
Includes time to modify source, synthesize, and test. Excludes approximately 2-3 days to read relevant documentation.

State Machine Learning vs HLS

Table 1 presents results from our empirical evaluation of AUTOMATASYNTH, including several metrics such as query counts and overall runtime. The penultimate column reports successful synthesis runs (✓), approximate results, and failures (✘) for AUTOMATASYNTH. HLS successfully synthesizes the same set of benchmarks. We report the approximate development time needed for an experienced developer to complete our standardized HLS workflow protocol.

AUTOMATASYNTH correctly learned thirteen of the eighteen benchmarks. Two benchmarks yield approximate solutions, with many of these approximations being extremely similar to the target kernel functionality. In our evaluation, this approximation was always the result of timeouts rather than the relative completeness of the SMT solver used for termination queries. The three unsupported kernels use computation that is difficult to capture with present string decision procedures.

We successfully synthesized hardware for the same fifteen benchmark kernels within the set time limit of six hours with HLS. The three we could not complete in time required heavy pointer arithmetic, a challenging function (`strtod`) to write for synthesis, and a significant amount of effort to generate suitable testbenches. We found that string kernels over unbounded inputs required significant effort to synthesize and verify with HLS. Benchmark functions involving multiple, non-sequential data accesses needed significant refactoring to allow for strictly linear (sequential) access with HLS stream abstractions. AUTOMATASYNTH, by contrast, has general support for *random access* at the source level—our framework automatically serializes this behavior to an equivalent DFA representation.

For HLS, we estimate a 2-3 day cost (beyond initial training) to read documentation to understand the limitations of HLS, the streaming functionality to allow for unbounded array accesses, and the requirements for synthesizable functions. Even without this time, HLS development took longer than AUTOMATASYNTH for nine of these benchmarks. Thus, AUTOMATASYNTH can significantly reduce the time burden on developers porting string functions to FPGAs.

Hardware Acceleration Performance

For spatially-reconfigurable architectures akin to FPGAs, the dominant factor affecting performance is the number of hardware resources used by a design, as well as the maximum clock frequency that the design can sustain, or F_{max}.

We use Grapefruit [3] to target the Xilinx

U280 FPGA with the automata generated by AUTOMATASYNTH, and used the same compilation script for our HLS results. To minimize the complexity of our comparisons, we performed an out-of-context evaluation and synthesized and place-and-routed the kernels on the FPGA.

Figure 3 shows the relative amount of hardware resources required by each string kernel as well as the maximum clock frequency that each design could sustain normalized to HLS results. Higher is better: bars with positive values mean that AUTOMATASYNTH used fewer resources (or has higher clock frequency) than the equivalent kernel implemented with HLS. On average, AUTOMATASYNTH produced hardware descriptions requiring 67.3% fewer Flip-Flops and 40.1% fewer LUTs, while processing data 88.7% faster than HLS. Part of this performance results from Grapefruit, which produces a highly tuned, RTL implementations for finite automata processing. Further, AUTOMATASYNTH learns the behavior of the source function rather than transforming the CFG directly. Thus, our approach is able to generate efficient state machines in spite of any inherent inefficiencies in the source code.

CONCLUSIONS

We present AUTOMATASYNTH, an end-to-end framework for accelerating Boolean string kernel functions using FPGAs. Our approach uses a novel combination of state machine learning, software verification, string decision procedures, and high-performance automata processing architectures to learn the behavior of a program and construct a behaviorally-equivalent FPGA hardware description. AUTOMATASYNTH successfully constructs equivalent (or near equivalent) FPGA designs for 83% of these benchmarks mined from open-source projects on GitHub. While modern HLS toolchain also supported 83% of these same benchmarks given similar development time, many kernels required significant hardware-specific modifications. Hardware generated by AUTOMATASYNTH also consistently outperforms HLS-generated hardware in terms of resource utilization and maximum clock frequency. We believe this approach shows promise for overcoming some of the limitations of current HLS techniques and easing the burden placed on developers.

ACKNOWLEDGMENT

This work is funded in part by: NSF grants CCF-1629450, CCF-1763674, CCF-1908633; AFRL Contract No. FA8750-19-1-0501; the Jefferson Scholars Foundation; and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

1. R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.
2. H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 35:1–35:12.
3. R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An open-source, full-stack, and customizable automata processing on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 138–147.
4. D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, Nov. 1987.
5. A. Biere, "Bounded model checking," in *Handbook of Satisfiability*, 2009, pp. 457–481.
6. K. Angstadt, J.-B. Jeannin, and W. Weimer, "Accelerating legacy string kernels via bounded automata learning," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 235–249.
7. G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus, "Debugging temporal specifications with concept analysis," in *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003, pp. 182–195.
8. F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, no. 2, pp. 86–95, Jan. 2017.
9. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Verifying web applications using bounded

model checking," in *International Conference on Dependable Systems and Networks*. IEEE, 2004, pp. 199–208.

10. B. Collie and M. P. O'Boyle, "Program lifting using gray-box behavior," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2021, pp. 60–74.
11. S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, May 2019.
12. D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190.

Kevin Angstadt, is an Assistant Professor of Computer Science at St. Lawrence University. Kevin received a BS from St. Lawrence University in 2014, a Masters degree from the University of Virginia in 2016, and PhD in computer science and engineering from the University of Michigan in 2020. His research focuses on improving programming support for emerging hardware technologies, including both the development of new programming models as well as automated techniques for adapting existing software. Contact him at kangstadt@stlawu.edu.

Tommy Tracy II is a Research Scientist in the Computer Science Department at the University of Virginia (UVa). He received his B.S. in 2010, M.E. in 2014, and Ph.D in 2019 all from UVa. His research focuses on network processing and accelerating pattern matching on FPGAs. Contact him at tjt7a@virginia.edu.

Kevin Skadron is the Harry Douglas Forsyth Professor of Computer Science at the University of Virginia, where he has been on the faculty since 1999, after receiving his PhD at Princeton. He served as department chair from 2012-2021. He is also director of the Center for Research on Intelligent Storage and Processing in Memory, part of the SRC JUMP program. He is a Fellow of the IEEE and the ACM, and a recipient of the 2011 ACM SIGARCH Maurice Wilkes Award. Skadron's research interests include design and application of accelerators and heterogeneous architectures, their memory hierarchies, and associated power, thermal, reliability, and programming challenges. He and his colleagues and students have developed a number of tools to support research on these topics, such as MNCaRT, HotSpot and Rodinia. Contact him at skadron@virginia.edu.

Jean-Baptiste Jeannin is an Assistant Professor of Aerospace Engineering at the University of Michigan. Jean-Baptiste received an Engineering degree from École polytechnique, France, and a Ph.D. in Computer Science from Cornell University in 2013. His research focuses on providing strong formal guarantees to aerospace and cyber-physical systems, using techniques from programming languages and formal verification. Contact him at jeannin@umich.edu.

Westley Weimer is a Professor of Computer Science and Engineering at the University of Michigan. Westley received a BA from Cornell and a Master's Degree and Ph.D. from Berkeley in 2005. His research focuses on static and dynamic analyses to improve software quality, particularly focusing on automatic or minimally-guided techniques that can scale and be applied easily to large, existing programs. In addition, he uses medical imaging techniques to study human interactions in software engineering. Contact him at weimerw@umich.edu.