IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 19 December 2024, accepted 5 February 2025, date of publication 10 February 2025, date of current version 14 February 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3540320

TOPICAL REVIEW

High-Level Synthesis for FPGAs—A Hardware Engineer's Perspective

SAKARI LAHTI AND TIMO D. HÄMÄLÄINEN, (Member, IEEE)
Faculty of Information Technology and Communication Sciences, Tampere University, 33720 Tampere, Finland
Corresponding author: Sakari Lahti (sakari.lahti@tuni.fi)

ABSTRACT The recent decades have witnessed unprecedented advances in the complexity of digital hardware systems, yet their design methods are still mostly based on manual register-transfer level (mRTL) languages such as VHDL and Verilog, introduced in the 1980s. While allowing exact system description, these languages have low productivity and require special expertise. High-level synthesis (HLS) promises to increase the productivity of hardware design by allowing system description from abstract, timeless source code, which is synthesized into optimized RTL code by an HLS tool according to technological constraints. However, HLS is still seen as somewhat immature technology with a non-consolidated offering of tools with varying features. Furthermore, the quality of results (QoR) of HLS is seen to be worse than with mRTL methods. This study sheds light on the status of HLS today. The emphasis is on field-programmable gate arrays (FPGAs) that allow fast development cycles. The study briefly covers the history of HLS, describes the HLS design flow, and lists the benefits and remaining challenges. The offering of current commercial and academic HLS tools is surveyed along with their features. A literature survey covering academic articles published between 2017 and 2024 on the QoR and productivity of HLS is presented. The results show that a gap of some margin still exists between the QoR of the HLS and mRTL methods. However, in productivity, HLS clearly outcompetes mRTL. Based on the study, several recommendations are made for HLS tool developers to close the QoR gap and accelerate the adoption of the method.

INDEX TERMS Design tools, field programmable gate arrays, high-level synthesis, productivity, quality-of-results.

I. INTRODUCTION

For decades, the efforts of the semiconductor industry have upheld Moore's law, "cramming more components onto integrated circuits." The Moore era may now have ended, but the industry has still found ways to increase the complexity of digital circuits [1]. This, in turn, has meant that a corresponding increase in design production volume has been required. While this can be achieved by hiring more engineers, the more effective way is to increase productivity, i.e., useful work done per hour per engineer. One of the primary ways for this has been to raise the abstraction level of the designs, allowing a design entry described in less detail, while electronic design automation (EDA) tools take care of the low-level features.

The associate editor coordinating the review of this manuscript and approving it for publication was Gian Domenico Licciardo.

In the 1980s, the introduction of hardware description languages (HDLs), Verilog, and VHDL, and more advanced EDA tools increased abstraction to the register-transfer level (RTL). In RTL, the system is described as a sequence of register transfers, where data are transformed from one state-saving register to the next. Despite being four decades old, the RTL paradigm is still dominant in the semiconductor industry. The HDLs have seen relatively minor updates over the years, with the introduction of SystemVerilog as the successor to Verilog being the largest. Meanwhile, language development has progressed rapidly in the software domain, with more expressive, safe, and robust languages coming to the fore.

High-level synthesis (HLS), also previously called behavioral synthesis, promises to be the next step in increasing the abstraction level of hardware design [2], [3], [4]. What counts as HLS somewhat varies from source to source, but

this work uses the definition from [5]: "HLS tools transform an untimed (or partially timed) high-level specification into a fully timed [RTL] implementation", a definition also supported by, e.g., [6]. To be more exact, HLS can be defined as a digital hardware design process where the system is described using a high-level, general-purpose programming language without the designer having to insert timing or micro-architecture into the description explicitly. Instead, the designer provides a timeless, behavioral description of the system and separately specifies technology constraints, such as the target platform and clock frequency. The HLS tool analyzes the algorithm, allocates necessary hardware resources, and schedules the required operations on different clock cycles. The output is an automatically generated RTL description of the system in VHDL or Verilog format.

One can take two views on the benefits of HLS: The hardware engineer's view or the software engineer's view [7]. The hardware engineer would like to use HLS to increase their productivity. Compared to manual RTL (mRTL) methods, time should be saved by omitting the tedious task of describing the microarchitecture of the system and designing state machines for the data path. HLS can also speed up verification by allowing high-level test benches and faster simulations. Furthermore, exploring the microarchitecture trade-offs and re-targeting of new platforms is straightforward. The software engineer, on the other hand, would like to accelerate all or some parts of their program on a field-programmable gate array (FPGA) without having to learn HDLs. For example, compute-intensive loops could be parallelized on the FPGA while a connected CPU manages IO and control flow.

Unfortunately, HLS is seen to have weaknesses for both these target groups [7], [8], [9]. The quality of results (QoR), mainly the performance and area consumption of HLS-designed applications, has been perceived as worse than with mRTL methods, which is undesirable for the hardware engineer [6], [7], [10]. In addition, a pure algorithm written with software mindset is rarely a good match for HLS tools [8], [11]. The designer has to modify the code to use bit-accurate data types, employ hardware design hierarchy, insert loop pipelining and unrolling, and consider the very different memory hierarchies between a CPU based on von Neumann architecture and an FPGA chip [12], [13], [14]. These modifications require skills that most software engineers lack, and they take time in any case.

This paper considers HLS from the hardware engineer's point of view. The aim is to shed light on whether HLS as a methodology is ready for widespread adoption in the digital hardware industry, where mRTL methods still dominate [15]. One of three possible results is expected when such a question is posed: 1) HLS is not mature enough to be of much use for a hardware engineer, 2) HLS can be used to augment mRTL flows and for fast prototyping, or 3) HLS is ready to replace mRTL methods as the primary hardware design technique. The answer to this question should mainly depend on the QoR and productivity of the HLS versus mRTL methods.

One of the contributions of this work is a literature survey that covers academic papers published between 2017 and 2024 to determine the difference in QoR and productivity between HLS and mRTL methods. As such, this paper can be seen as a continuation of our earlier work, which examined the same question by reviewing articles published between 2010 and 2016 [16]. We also provide a brief survey of currently available HLS tools along with their main features, list the main benefits and challenges of HLS, and provide recommendations for HLS tool developers based on the findings.

The work is restricted to studying HLS on FPGA platforms. FPGAs are inexpensive to obtain in small quantities compared to application-specific integrated circuits (ASICs) that require expensive manufacturing processes at IC foundries. Furthermore, unlike ASICs, they are re-programmable, which makes them ideal for rapid prototyping cycles. For these reasons, most academic literature on HLS focuses on targeting FPGA platforms, and few studies on HLS on ASIC can be found. However, it can be assumed that most of the results obtained for FPGAs can be generalized to ASICs. For example, the productivity benefits of generating RTL files for an FPGA project should be similar to those of an ASIC project. The relative QoR differences also depend on the generated RTL, and there is no obvious reason to assume that using an ASIC technology library would hinder an HLS tool compared to an FPGA technology library. That said, this work does not make explicit claims about using HLS with ASIC flows.

HLS has been an area of active research during the last two decades. For example, the IEEE Xplore database provides 4,566 results with the search "high-level synthesis" between the years 2004 and 2024, 3,935 results between 2014 and 2024, and 2,434 results between 2019 and 2024, showing a constantly growing interest. Some prominent papers that provide a wide view of HLS can be recognized. Cong et al. survey HLS success stories from the last decade and identify some remaining challenges and areas for further research [6]. Sozzo et al. compare HLS with other methods to program FPGAs and identify their benefits and drawbacks [15]. Molina et al. survey HLS-related computing models, methodologies, and frameworks comparing their features and limitations [17]. Schafer and Wang survey DSE in HLS, observing related opportunities and challenges [18]. Furthermore, several surveys on existing HLS tools and frameworks can be found [8], [19], [20], [21]. However, no previous work covers multiple aspects of HLS simultaneously, including tools, QoR, productivity, and challenges, and draws holistic conclusions based on all of these, which is the main contribution of this paper.

The remainder of this paper is divided as follows. Section II provides a brief introduction to HLS, including the history of

the method. Section III describes the HLS design flow from both the user's and tool's point of view. Section IV lists the benefits and challenges of HLS compared to mRTL methods as seen in the previous literature, and Section V surveys the HLS tools currently available and their features. Section VI contains the wide literature survey on QoR and productivity of HLS. Finally, Section VII discusses the findings and provides recommendations for the further development of the HLS tools, and Section VIII concludes the work.

II. HLS AS A HARDWARE DESCRIPTION METHOD

Although traditional HDLs have dominated FPGA design entry generation since the 1980s, viable competitors have appeared to boost productivity during the last 20 years or so. These include not only HLS but also modern HDLs and domain-specific languages (DSLs) [15].

Modern HDLs strive to increase the abstraction level from traditional HDLs and import features from modern software programming languages. Some are based on functional programming languages (e.g., Clash [22], Chisel [23]), and some on imperative languages (ArchHDL [24], PyVerilog [25]). In contrast, others are extensions of SystemVerilog (BlueSpec [26], TL-Verilog [27]). They come with tools that transform the description into VHDL or Verilog, as synthesis programs do not directly support modern HDLs. However, they still retain the nature of traditional HDLs by relying on timed and architecture-oriented design and, therefore, are not counted as HLS.

DSLs provide a narrower focus, yielding a highly optimized design flow for a target domain. This is their strength, but also their weakness, as the narrow focus limits their broader usability. DSLs can be divided into those that focus on a specific application domain (e.g., Hipacc [28], Halide-HLS [29]), architectural domain (SuSy [30], Spatial [31]) and those that provide an intermediate layer between a DSL and RTL (HeteroCL [32], AnyHLS [33].

This work is concerned with HLS. Whereas the previous methods primarily aim at hardware engineers to increase their productivity, HLS also allows software engineers to accelerate parts of their code on FPGA. This is because HLS tools use input languages familiar from the software domain. Another key difference from HDLs is that HLS is at least partially untimed, that is, there is no explicit clock or other concept of time in the input code. However, some HLS tools do accept partially timed input languages, such as SystemC. In this study, only tools that allow untimed input, at least as an option, are considered HLS. Tools that require describing the microarchitectural layout of the system are also considered non-HLS, as that would make them HDL tools by definition (even if the source code is in a traditional programming language). A general-purpose HLS tool must also be domain independent, unlike DSLs.

The history of HLS can be divided into generations as Martin and Smith do in [10]: Generation 0 (1970s) is the prehistory of HLS. The early research laid theoretical

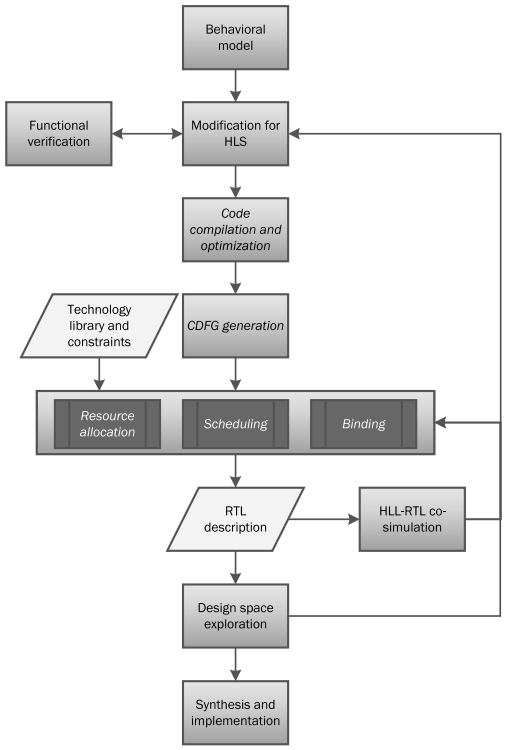

FIGURE 1. HLS design flow. Steps performed by the HLS tool are in italics.

foundations, but this was before the VLSI revolution, and the EDA technology of the time was not up to practical applications. Generation 1 (1980s to early 1990s) continued to create the research foundation of HLS. The first HLS tools became available, but were ignored by the industry that was adopting synthesis with HDLs. Generation 2 (mid-1990s to early 2000s) saw the first HLS tools provided by the major EDA companies. This generation also failed to gain traction due to low QoR, obscure input languages, and the focus on data flow-oriented applications. A significant expansion of HLS tool vendors characterized Generation 3 (mid-2000s onward). This was the first generation that saw commercial success due to improved QoR, the adoption of widely used input languages such as C++ and MATLAB, and the spread of FPGAs, which are a good fit for application development with HLS.

Sozzo et al. view us as residing among the fourth generation of HLS tools [15]. They define this generation by the fact that tool vendors now integrate intellectual property (IP) component development with system-level design flows. The IPs are designed with HLS and connected with each other, CPUs, and memories using an integrated design flow. These fourth-generation tools are called accelerator-centric synthesis (ACS) tools. However, most current HLS tools, as surveyed in Section V, do not belong to this category. Moreover, since most ACS tools are tied to vendor-specific flows, they will not completely replace the more generic third-generation tools.

III. HLS DESIGN FLOW

Fig. 1 shows in detail the FPGA design flow for HLS. The flow starts from the behavioral model written for CPU simulation, which unfortunately cannot be used directly as input to the HLS tool even if the language it uses is supported. The QoR would be poor because of the differences in hardware architecture between a typical CPU system and an FPGA. For example, the clock speed of an FPGA is typically an order of magnitude lower than that of a desktop CPU system. On the other hand, FPGA allows massive parallelization of computations, whereas CPUs execute programs sequentially. The memory architecture is also quite different.

Typical required modifications include:

- Removal of dynamic memory handling, recursion, and function pointers as HLS does not support them,
- Conversion of native integer and floating point data types to bit-accurate fixed point data types,
- Division into hierarchical components with streaming communication channels to enable coarse-grained pipelining,
- Interleaving, merging, and widening of arrays mapped to memories to enable multiple data access,
- Implementing explicit caching of data fetched from off-chip memories and,
- Replacing mathematical functions, such as division, square root, trigonometric functions, and absolute value, with iterative and approximative versions more suitable for FPGAs.

After all these modifications, the algorithm's IO behavior should remain unchanged compared to the original model. This requires implementing a functional verification testbench, which sends the same input data to both the original and the modified models and checks that the output remains the same. With some algorithms, this is not possible when, for example, floating-point data types are transformed into more inaccurate fixed-point data types. Tolerances that can be tested with, e.g., signal-to-noise ratios, must be set in these cases. [14] et al. provide a broader overview of the transformations required for high QoR in HLS.

The verified source code can now be sent to the HLS tool's internal analysis flow. The source code is compiled and optimized, which can be done with any compiler designed for the source language. The optimizations performed by the compiler can include, for example, constant propagation, dead-code elimination, and partial-redundancy elimination [34]. The output is an intermediate representation (IR) further analyzed by the HLS tool in the control and dataflow graph (CDFG) generation step. The CDFG is a directed graph whose edges represent the algorithm's control flow (i.e., conditional branches) and nodes the operations without branches [5]. The CDFG lists the algorithm's arithmetic, logical, and data movement operations and shows their possible execution orders. Thus, some operations can be done in parallel, whereas others must be sequential as they depend on the results of the previous ones. An example CDFG for HLS can be found in [3].

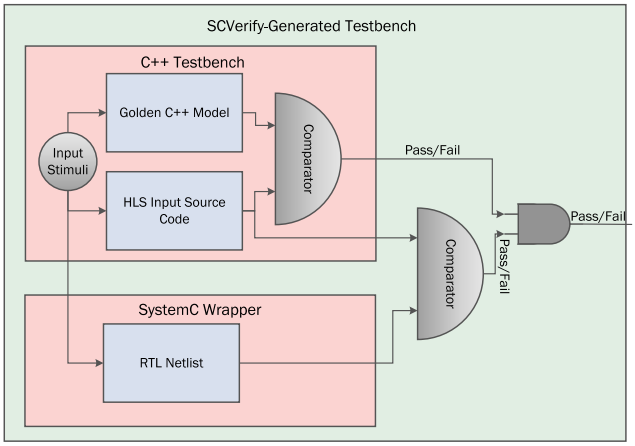

FIGURE 2. SCVerify infrastructure.

In the next step of the flow, the user must specify the target technology and primary design constraints, the most important being the clock frequency. The technology is represented by a library that characterizes a particular FPGA chip (or ASIC technology). It contains a list of the basic arithmetic, control, logic, and storage operations available, their cost in resources or area, and their delay and power consumption characteristics. Memory models can also be supplied at this stage. In addition, the provided design constraints can include mapping of IO to desired resources (e.g., wire-type interfaces or FIFOs) and loop transformations (e.g., unrolling and pipelining). The HLS tool then performs the resource allocation, scheduling, and binding steps based on the target technology, design constraints, and input source code.

In the resource allocation step, the HLS tool selects a hardware component from the technology library for each operation in the CDFG. This can be an automated process or guided by the user. The selected components are annotated with area and delay information. Time is inserted into the system in the scheduling step by placing the operations on different clock cycles according to the dependencies in the CDFG, clock period, and selected loop transformation options. An operation might take several clock cycles to complete or just one. A state machine that controls the data path is automatically generated according to the schedule. The binding step combines resource allocation and scheduling results, selecting an appropriate hardware resource for each operation on each clock cycle. It also maps variables to storage elements and data transfers to buses. Many HLS tools perform the allocation, scheduling, and binding steps in this order, but some may order them differently according to their optimization algorithms [5].

The output of the previous steps is the RTL description of the system in VHDL or Verilog (or both). The produced RTL should match the untimed input model's IO behavior one-to-one, but there are cases when they may differ. For

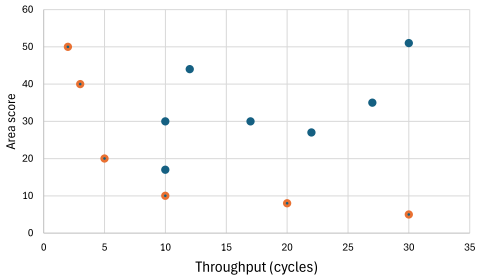

FIGURE 3. A Pareto front for area/throughput (orange dots). Solutions on the front dominate all other solutions, at least in one of the metrics. For each solution not on the Pareto front, there is at least one solution on the front, which dominates it on both metrics.

example, suppose that there is a feedback path on the data path. In the untimed behavioral model, the feedback data is available immediately, whereas in RTL, it may have several clock cycles of latency, affecting the results. HLS tools often provide a verification environment that pits the untimed model and the RTL against each other and checks that they produce the same output with the same inputs. For example, the Catapult HLS tool [35] enables this with its SCVerify environment, shown in Fig. 2. A SystemC wrapper is automatically generated that feeds the input from the C++ testbench to the RTL netlist simulation. The output is compared to the output from the C++ source code to determine whether the models are equivalent.

If there is a mismatch between the input source code and the RTL models in simulation, the cause should be identified and fixed in either the input source or the architectural constraints. Otherwise, the produced RTL can be studied for quality metrics such as resource and power consumption and throughput. Usually, the QoR is not close to optimal on the first synthesis pass, which necessitates design space exploration (DSE) by inspecting different architectural choices, such as loop pipelining and unrolling options and memory architecture solutions. The goal is to find a solution on the Pareto front with respect to the QoR parameters (Fig. 3). In practice, this usually means studying the trade-offs between performance and area. Often, it is enough to change code pragmas in the input source or GUI options within the HLS tool when performing DSE, but sometimes more extensive changes to the source code need to be made. Once a suitable solution has been found, the design flow can continue to synthesis and implementation with the downstream tools. Efficient DSE in HLS is an area of active research. The interested reader can refer to, for example, [17], [18], or [36].

TABLE 1. Benefits and challenges of HLS.

Benefits	Challenges
Reduced effort in design entry	Possibly lower QoR
Improved verification process	Complications in verification
Enables DSE	Difficulty of incremental changes
Easier reuse of designs	Lack of standardization
Leveraging of software techniques	Limitations of the input languages
Familiar input languages	Time-consuming DSE
	Lack of trained engineers

IV. BENEFITS AND CHALLENGES OF HLS

In the previous literature, many benefits and challenges compared to mRTL methods have been attributed to HLS. Table 1 lists the most important ones. A discussion of the items follows, with benefits first.

A. REDUCED EFFORT IN DESIGN ENTRY

The most obvious benefit in HLS is the reduced design effort thanks to the higher abstraction level. The three main aspects of higher abstraction compared to mRTL are related to timing, data, and interface synthesis [3]. Time is not present in the HLS source code but is inserted by the HLS tool based on synthesis constraints. Most HLS tools also support high-level abstract data types, classes, and structs familiar from programming languages. Interfaces can be synthesized from pre-made libraries instead of being expressed explicitly in the source code. The mapping of variables to memories or registers and operations to different implementations of functional resources is also absent from the source code, allowing a higher abstraction level [2]. Design productivity improvements on the order of 2x can be expected with HLS [3], but this is studied further in Section VI.

B. IMPROVED VERIFICATION PROCESS

HLS also improves the verification process in many ways. First, the higher abstraction level leads to fewer errors related to manual coding of architectural details [3], [5]. Second, verification can start earlier in the design process without waiting for the mRTL coding [3]. Third, simulating code on a higher abstraction level enables faster simulation speed. The difference can be orders of magnitude [2].

C. ENABLES DSE

In HLS, microarchitecture is not expressed in the source code but implemented by the HLS tool. This allows directing the microarchitecture synthesis by code pragmas and knobs, allowing efficient DSE to evaluate their effect on performance, area, and power. The HLS tool is also free to analyze the CDFG created from the input code using various transformations to find optimal solutions. This contrasts with mRTL, where the designer has to manually create a fixed state machine and allocate resources [7].

D. EASIER REUSE OF DESIGNS

The higher abstraction level also allows design reuse to be more efficient for different target technologies or performance goals [2]. This often does not require changes to the input source code, as the HLS tool handles microarchitecture-related details.

E. LEVERAGING OF SOFTWARE TECHNIQUES

The HLS source code is based on software programming languages. As such, software optimization techniques can be leveraged with HLS. These include various compiler optimizations and transformations for higher-quality hardware

and software debugging tools for verification [7]. For example, LLVM-based compiler frameworks [37] can be used to parse the source code into an intermediary representation to take advantage of a rich set of optimization tools [3].

F. FAMILIAR INPUT LANGUAGES

HLS uses familiar languages from the software domain for input, and it is still a fact that more engineers know these languages instead of HDLs. This lowers the threshold for adopting HLS compared to the mRTL techniques. In addition, many systems involve both software and hardware components, where it is helpful to have a common input language for both domains [38].

The following challenges are attributed to HLS.

G. POSSIBLY LOWER QOR

Perhaps the most common problem attributed to HLS is the perceived lower quality of the results compared to mRTL. A significant gap is claimed to exist between the achievable clock frequency of HLS and mRTL designs, which favors mRTL [6]. This is ascribed to the interconnect delays that HLS struggles to predict and optimize [9]. Another reason is the expertise required to write quality code for HLS and efficiently use the pragmas directing micro-architecture generation.

H. COMPLICATIONS IN VERIFICATION

The design entry in HLS is done at a high level, but the output of the tool is RTL code. The simulations for the high-level code run faster, but the RTL output should also be verified to be correct, as translation errors may occur [39]. This requires either slower RTL simulations or formal proof of the equivalence between the source and the output [3]. Research on formal equivalence checking is active but inconclusive [40], [41], [42], [43].

I. DIFFICULTY IN INCREMENTAL CHANGES

Often small changes are required to a design after it has been placed and routed. A problem with HLS is that it is difficult to find a correlation between the abstract source code and the details of the implemented design. Furthermore, engineering change orders (ECOs) are often needed in ASIC designs to change them incrementally toward the end of a project. This requires incremental synthesis, which is a complicated process for the HLS tools [3].

J. LACK OF STANDARDIZATION

There are few to no industry-wide standards on HLS. The compatibility between the tools is low since they have significant differences between the supported features, libraries, and core abstractions [6]. This is in contrast to mRTL methods, where VHDL and SystemVerilog standards enable the use of the same source code from the synthesis tool to the synthesis tool with few modifications needed.

K. LIMITATIONS OF THE INPUT LANGUAGES

The usage of high-level input languages also poses some problems for HLS [3], [38]. Designing clock-accurate systems is challenging when there is no explicit concept of time in the source code [44]. Many languages, such as C++, also lack the notion of concurrency, so the designer has to express parallel constructs in some other way using the HLS tool. In addition, native data types are usually too limited to convey signals of arbitrary bit widths. The memory model of CPUs is also quite different from FPGA or ASIC architectures, which can contain many small memories. Furthermore, legacy code targeting software architectures usually contains non-synthesizable constructs, such as dynamic memory handling, recursion, exception handling, and multi-threading, which must be removed for HLS [6].

L. TIME-CONSUMING DSE

While HLS enables faster DSE than HLS, it can still be a time-consuming endeavor [9], [17], [18]. The design space is usually substantial and multidimensional as at least area, performance, and power must be optimized simultaneously. HLS DSE can be performed by changing global synthesis options, tuning microarchitecture-related options, or changing the source code. These modifications take time, and the following synthesis runs consume it as well.

M. LACK OF TRAINED ENGINEERS

The final noteworthy challenge with HLS is the lack of trained engineers [3]. Acquiring a high QoR with HLS requires specific knowledge and skills [6]. Software engineers are commonplace, but current HLS tools require hardware expertise that they usually lack. Hardware engineers are less common, and most are trained in mRTL methods instead of HLS.

V. CURRENT HLS TOOLS

This section summarizes existing HLS tools, both academic and commercial. Several previous tool surveys can be found in the literature, for example, [8], [15], [19], [20], [21]. However, the evolution of the tools is rapid. To illustrate this point, the number of commercial tools available decreased from 13 to 9 between the surveys of 2016 ([20]) and 2020 ([8]). This section aims to determine whether this HLS tool market consolidation has progressed further. It also surveys the input and output languages that the tools use, what kind of target platforms are supported, and makes other observations about the tools.

A. LIST OF CURRENT HLS TOOLS

A brief survey of academic and commercial HLS tools that have ongoing or recent support follows. Tools that have not received updates since 2020 are considered abandoned and are omitted. The survey is based on Web searches and previous surveys. Some HLS tools may have been missed because references to them are difficult to find. Such

TABLE 2. List of current HLS tools.

Tool	Owner	License	Input	Output	Targets
Bambu	Politecnico di Milano	Acad.	C, C++	Verilog, VHDL	Selected FPGAs and ASICs
Catapult	Siemens	Comm.	C, C++, SystemC	Verilog, VHDL	Most FPGAs and ASICs
CyberWorkBench	NEC	Comm.	C, C++, SystemC	Verilog, VHDL	AMD, Intel FPGAs and ASICs
Dynamatic	EPFL	Acad.	C, C++	Verilog, VHDL	AMD FPGAs
HDL Coder	MathWorks	Comm.	MATLAB, Simulink models	Verilog, VHDL, SystemC, SystemVerilog	AMD, Intel, and Microchip FPGAs, and ASICs
HLS Compiler	Intel	Comm.	C++	Verilog	Intel FPGAs
Kiwi Compiler	Microsoft, University of Cambridge	Acad.	C#	Verilog	AMD, Intel FPGAs
oneAPI	Intel	Comm.	C++, SYCL	Verilog	Most FPGAs
SmartHLS	Microchip	Comm.	C, C++	Verilog	Microchip FPGAs
Stratus	Cadence	Comm.	C, C++, SystemC, MATLAB	Verilog	All FPGAs, ASICs
Vitis	AMD	Comm.	C, C++	Verilog, VHDL	AMD FPGAs
XLS	Google	Comm.	DSLX	Verilog	Most FPGAs, ASICs

tools would likely be academic tools with few publications related to them. A comparative evaluation of user experience, productivity, and QoR between the different HLS tools would be of interest, but goes beyond the scope of this study.

As the hardware engineer's point of view of HLS is taken here, tools that are only aimed at software developers to accelerate parts of their code on FPGA are omitted (e.g., Hastlayer [45] and CacheQ [46]). HLS tools developed as a hobby project or without access information are also ignored in this survey (e.g., PipelineC [47] and Kanagawa [48]). Domain-specific HLS tools are not included either (e.g., OpenHLS [49] for deep neural networks or Hipacc [50] for image processing kernels) since the focus is on tools that can replace mRTL in all application domains. Finally, the so-called HLS code generation tools are omitted, as they do not provide RTL output, but code intended as input for actual HLS tools (e.g., Hot & Spicy [51]).

Table 2 lists the tools found in the survey that satisfy the previous criteria. The tool's owner, license type, input and output languages, and supported target platforms are listed. Further details and observations are given in the tool-specific paragraphs.

1) BAMBU

Bambu [52] is an open source HLS tool, part of the PandA hardware-software co-design framework by Politecnico di Milano. Its inputs are a C/C++ source code file and an XML configuration file, and the output is available in Verilog and VHDL. Bambu has an automated bit-width analysis tool to reduce register area, and it can automatically generate a testbench for C/RTL co-simulation to validate that the generated RTL is functionally equivalent to the source code. Selected AMD, Intel, and Lattice FPGAs are available as synthesis targets, and a few OpenROAD [53] ASIC libraries.

2) CATAPULT

The Catapult HLS [35] by Siemens can target both AMD and Intel FPGAs and various ASIC libraries based on the C++ or SystemC input, producing Verilog and VHDL output. Code-inserted pragmas and GUI options can specify parallelism, throughput, and memory architecture. Additional features include a value range analysis tool for fixed-point variable width optimization, a Gantt chart schedule viewer, cross-probing between source code and generated RTL, and power optimization. Verification is supported by source/RTL co-simulation, a coverage metric tool, and formal equivalence checking.

3) CYBERWORKBENCH

The NEC CyberWorkBench HLS [54] accepts C/C++ and SystemC input and produces Verilog and VHDL output. It can target any ASIC technology and AMD and Intel FPGAs. The tool supports power optimization, multiple clock domains, a C property checker, and C/RTL cosimulation. However, the current support for CyberWorkBench is uncertain, as the most recent documentation available for the software on its Web page is from 2018.

4) DYNAMATIC

An academic, open-source HLS tool developed at EPFL, Dynamatic [55] is based on the MLIR compiler infrastructure. It takes a C/C++ input and can produce a Verilog and VHDL output. The generated RTL takes the form of a dataflow circuit. Only AMD FPGAs are supported as synthesis targets. A unique feature of Dynamatic is the ability to generate dynamically scheduled circuits. Compared to static scheduling, this can increase throughput in cases where the taken data path operations depend on run-time control decisions.

5) HDL CODER

The HDL Coder [56] by MathWorks takes MATLAB functions and Simulink[1] models as input and produces Verilog, SystemVerilog, and VHDL output that can target AMD, Intel, and Microchip FPGAs and ASIC libraries. The HDL Coder

[1] Simulink is not considered HLS in this work as it uses a model-based design flow.

can also generate SystemC output, which is designed to serve as input to the Cadence Stratus HLS design flow. The HDL Coder design suite includes tools for converting MATLAB floating-point values to hardware-friendly fixed-point values, MATLAB/RTL functional equivalence checking, and libraries for common functions in deep learning, wireless communications, and signal processing domains.

6) HLS COMPILER

The Intel HLS compiler [57], included in the Quartus Prime Design Software Pro Edition installation, takes untimed C++ as input and produces RTL Verilog as output. Only Intel FPGAs can be targeted for synthesis. C++/RTL co-simulation is supported. The Intel HLS compiler is planned to be discontinued in 2024 [58] to be replaced by the fourth generation Intel oneAPI DPC++/C++ Compiler [59], which can target not only FPGAs but also CPU, GPU, and custom hardware systems.

7) KIWI COMPILER

The Kiwi Compiler [60] was jointly developed by the University of Cambridge Computer Laboratory and Microsoft Research Limited. Unique among HLS tools, it uses C# as its input language. The output is in Verilog, and AMD and Intel FPGAs are possible targets. Kiwi supports multi-dimensional arrays, threading, file-server IO, and limited recursion in its source code, which are usually forbidden by other HLS tools. Even more interestingly, it allows dynamic allocation of objects and object pointer manipulation that are ubiquitous in software programming but are banned by most HLS tools. The most recent update to the file repository was two years ago, and the status of the web pages suggests that the Kiwi Compiler may not receive updates in the future.

8) ONEAPI

The Intel oneAPI toolkit [59] allows to deploy code on CPUs, GPUs, and FPGAs based on a unified framework, which uses SYCL-based C++ language extensions. Maintained by the Khronos Group, SYCL is a cross-platform abstraction layer that allows algorithms to switch between target platforms without changing the source code [61]. OneAPI supports FPGA synthesis by allowing a C++ testbench-based RTL simulation on the Questa-Intel simulator. OneAPI is set to replace the HLS Compiler as the only HLS tool provided by Intel. The features of oneAPI make it a fourth-generation ACS tool.

9) SMARTHLS

Formerly known as LegUp [62], which originated as an open-source academic tool, it was acquired by Microchip and renamed SmartHLS [63]. SmartHLS accepts C/C++ as its input code and produces Verilog output meant to be synthesized with Microchip's Libero SoC design suite and programmed on a Microchip FPGA. Source/RTL co-simulation is supported with the Siemens ModelSim simulator [64].

10) STRATUS

The Cadence Stratus HLS [65] accepts as input C, C++, SystemC, and MATLAB functions and produces Verilog output. The tool is based on Forte Design System's Cynthesizer (acquired in 2014) and Cadence's own C-to-Silicon compiler. Both FPGAs and ASIC are targetable. Stratus ships with a library of commonly needed functions implemented in SystemC, such as connectivity interfaces, mathematical functions, and multi-clock design utilities. Source code/RTL co-simulation is supported, and the tool can also create ECO patches for existing designs.

11) VITIS

The AMD Xilinx Vitis HLS [66] is perhaps the most popular HLS tool, at least in academia (based on the survey in Section VI) thanks to the popularity of AMD FPGAs and Vitis shipping with their development tools. It was originally developed by AutoESL under the name AutoPilot but was acquired by Xilinx in 2011. It was renamed Vivado first and Vitis later. It accepts C/C++ as input language, produces Verilog and VHDL output, and can target only AMD FPGA devices. Vitis heavily depends on pragmas to enable parallelism and pipelining from the sequential source code and to control the memory architecture. C/RTL co-verification is supported. Vitis HLS is now part of the AMD Vitis Unified Software Platform, which can be used to create designs that target ARM processor subsystems, AI engines, and FPGAs [67]. This makes it a fourth-generation ACS tool.

12) XLS

XLS [68] is an open-source HLS tool developed by Google. It uses the Rust-like DSLX language to produce synthesizable designs for FPGAs and ASICs. XLS is described as a ''mid-level'' synthesis tool that provides an HLS-like experience and allows for controlling low-level details such as pipeline stages and worst-case throughput. At the same time, both software models and synthesizable (System)Verilog code can be generated from the same source. XLS is stated to be an experimental and rapidly developing project that Google does not officially support.

B. OBSERVATIONS ON CURRENT HLS TOOLS

Twelve existing HLS tools were found to satisfy the criteria defined at the beginning of the section. Nine of the tools are commercial, and three are academic. Intel HLS Compiler will be discontinued in 2024 in favor of Intel oneAPI, which brings the number of commercial tools to eight. Compared to the 2020 survey by Numan et al. [8], the number of commercial tools has decreased by one (by 11%) and the number of academic tools by four (57%). The number of commercial HLS tools provided appears to be stabilizing, but interest in academic tools is waning. As commercial tools

mature and become more complex, academia may see less room for providing attractive free alternatives.

All the tools except HDL Coder, Kiwi Compiler, and XLS support pragma-annotated C/C++ as an input language, with some tools supporting SystemC. The Kiwi Compiler supports C# and HDL Coder MATLAB functions and Simulink models, but these choices can be seen as the background companies' intentional support for languages developed by themselves. Tool vendors without vested interest in a specific language tend to favor C/C++, which was explored as an HLS language in [69].

Co-verification between the untimed source and generated RTL, based on the behavioral testbench, seems to be a common feature of modern HLS tools. Most of the surveyed tools mentioned support for this property. This is a positive state of affairs, as reusing the behavioral testbench for RTL verification saves verification effort in general and allows validation that the tool-generated RTL corresponds to the source code on a behavioral level.

All tools support Verilog as an output language; six of the twelve also support VHDL. The missing VHDL support in some tools is not a significant omission since practically all commercial synthesis tools accept Verilog input, and the HLS tool-generated output is not intended to be human-readable in any case. The target platform support is more varied. All tools provide support for FPGA synthesis, but it depends on the tool whether the support is vendor-independent or not. For example, Intel HLS Compiler, Microchip SmartHLS, and AMD Vitis provide support only for their parent company's FPGAs. Other commercial tools provide broader support, but academic tools are more limited, likely due to smaller developer resources. Finally, half the tools include ASIC synthesis support. Most of these tools are from vendors that do not manufacture FPGA chips, so the tools are intended to be more generic than only backing the company's FPGAs.

VI. LITERATURE SURVEY ON QOR AND PRODUCTIVITY OF HLS

This section is a continuation of [16] by Lahti et al., which explored the QoR and productivity gaps between HLS and mRTL methods based on a literary survey. That paper surveyed 46 scientific articles published between 2010 and 2016 that compared the development effort and QoR of applications developed with both HLS and mRTL methods. The conclusion was that, while manual methods tend to produce better performance and lower resource consumption scores than HLS, the increased productivity of HLS compensated for this, making the choice between the methods dependent on the priorities of a project.

This work extends that study. The range of surveyed years has been shifted from 2010-2016 to 2017-2024 to focus on the quality of HLS tools and methods during the last years and to see if any new trends are visible. The updated survey is based solely on the literature, whereas the older one was partially based on a test group study. However, the relatively inexperienced test group of students was not likely to reflect the actual ability to leverage either design method.

This updated survey aims to answer the following questions:

1) What is the difference in productivity between HLS and mRTL?

TABLE 3. List of surveyed papers.

[#]	Year	HLS Tools	Applications	# apps.
[70]	2017	Vivado	Discrete cosine transform	2
[71]	2017	Vivado	K means clustering: Lloyd's algorithm, filtering algorithm	2
[72]	2017	Vivado	Tiny encryption, blowfish encryption	2
[73]	2017	Vivado	Galois field multiplication and division	2
[74]	2017	Catapult	HEVC 2-D DCT/DST	1
[75]	2017	Vivado	Black-Scholes financial, Black 76 financial, binomial option pricing	3
[76]	2017	Vivado	PID control, YNK control, plant control	3
[77]	2018	Vivado	Light propagation simulation	1
[78]	2018	Catapult, Vivado	CCSDS 123.0-B-1 image compression	2
[79]	2018	Undisclosed	Euclidean distance, qfrom	2
[80]	2018	Vivado	Linearization, IIR filter	2
[81]	2018	Vivado	Packet parsing	2
[82]	2018	Vivado	Crystal identification, event sorter	2
[83]	2018	Vivado	MIMO detector	1
[84]	2018	LegUp	Floating point cores	2
[85]	2019	Vivado	RSA, SHA3-256, AES	3
[86]	2019	Vivado	Polar decoder	1
[87]	2019	Vivado	NTRUEncrypt	2
[88]	2019	Vivado	Robot OS component	1
[89]	2019	Vivado	PRESENT encryption	1
[90]	2020	Intel OpenCL	Heapsort	2
[91]	2020	HDL Coder	Digital down converter	1
[92]	2020	Vivado	Stencil computation	1
[93]	2020	Vivado	RADAR signal processing	1
[94]	2020	Vivado	Number theoretic transform	8
[95]	2020	Vivado	Sobel edge detection	1
[96]	2020	Vivado	Sobel edge detection	1
[97]	2020	Vivado	Key-value store	1
[98]	2020	Vivado	Principal component analysis	1
[99]	2020	Stratus	RISC-V processor	1
[100]	2020	Vivado	LeNet-5 CNN	1
[101]	2020	Vivado	Number theoretic transform	1
[102]	2021	HDL Coder	Harris corner detection	1
[103]	2021	Vivado	PRESENT, AES/Rijndael, Serpent encryption	3
[104]	2021	Catapult	HEVC interpolation filter	1
[105]	2021	Vivado	Kyber-512/513 encoders	2
[106]	2021	Vivado	PID control	1
[107]	2021	HDL Coder, Vivado	Buck converter, full-bridge converter	4
[108]	2022	Vivado	VVC fractional interpolation	1
[109]	2022	Vivado	Slew-rate reduction shaping function	1
[110]	2022	Vivado	Runge-Kutta fourth order	1
[111]	2022	Vivado	Network intrusion detection DNN, ResNet	2
[112]	2022	HDL Coder	AES 128	1
[113]	2022	Undisclosed, OneAPI	Sobel filter	2
[114]	2022	Undisclosed	Fixed priority selector, integer square root	2
[115]	2023	Vivado	ODFM channel estimation/equalizer, openwifi	2
[116]	2023	HLS Compiler	Moving average, triangular smooth, peak finder	3
[117]	2023	Bambu, MaxCompiler, Vivado, XLS	IDCT	4
[118]	2023	HDL Coder	MUSIC direction of arrival	1
[119]	2023	Vivado	Trading system order processing	1
[48]	2024	Kanagawa	Software-defined networking, Network-protocol engine, AES-256 GCM	3

2) What is the difference in achieved performance between HLS and mRTL?
3) What is the difference in resource consumption between HLS and mRTL?
4) Can differences in the previous be detected based on the HLS tool?

As an aside, the survey also observes the popularity of different HLS tools, at least in academia.

A. QUALIFYING PAPERS

Scientific peer-reviewed articles were searched from various online databases. These included IEEE Xplore, Google Scholar, ACM Digital Library, Scopus, Springer Link, asXiv.org, EBSCOhost, and Science Direct. Papers that included "high-level synthesis" in the title or abstract were searched and sorted by relevance. A paper was chosen for a closer study if the title or abstract suggested that it could contain comparisons between HLS and mRTL implementations of the same application. The searches returned thousands of papers, but thanks to the sorting, the density of interesting papers dropped enough to stop the inspection of a given database after a certain point. The search was restricted to articles published between 2017-2024.

A paper was accepted for inclusion in this survey if it met the following conditions: 1) It included a comparison of one or more FPGA applications implemented in both HLS and mRTL methods, and 2) it reported at least one of the following for both implementation methods:

- Development effort expressed in either development time or lines of code (LoC),
- performance expressed in execution time, latency, maximum clock frequency, or similar metric, or
- FPGA resource consumption.

Another interesting metric is power consumption, but too few papers reported on it to make reliable generalizations. This study also ignored ASIC implementations (and there were few in any case), except for development effort comparisons, as the paper topic is on FPGAs.

Some articles considered tools such as BlueSpec [120] or Chisel [23] to be HLS, but this work uses the definition of [15], which excludes such hardware description tools as well as domain-specific tools. Those articles were therefore rejected from the survey. Another common reason for the disqualification of a paper was that the synthesis based on HLS and mRTL was done on different FPGA chips. As chips vary in resources, implementation technology, and timing capabilities, this comparison is unfair. This was common in papers that compared their HLS implementation to a referenced mRTL solution. Otherwise, both the authors' self-made and referenced mRTL comparisons were accepted for this survey.

Table 3 shows a summary of the articles accepted in the survey. It lists the publication year, the HLS tools used, the algorithms implemented, and the number of applications in the paper. A summary of the data year by year can be found in Table 4. Fifty-one papers were found, enough for a numerical analysis of aggregate data. However, there are not enough papers year by year to draw reliable conclusions about possible trends that might emerge in the data over time on that resolution. The table also shows the number of applications in the qualified papers by year. An application is defined as either a separate algorithm or a variation of some relevant parameter. It was counted as its own application if the same algorithm was synthesized with different HLS tools, micro-architectural options, or on different FPGA platforms. Each such application became its own data point in this survey, as presented in the following sections. On average, each paper contained 1.8 different applications. Note that a single HLS-mRTL pair of the same application counts as one.

TABLE 4. Number of survey papers and applications by year.

Year	Papers	Applications
2017	7	15
2018	8	14
2019	5	8
2020	12	20
2021	6	12
2022	7	10
2023	5	11
2024	1	3
Total	51	93

TABLE 5. Metric occurrence frequency in reviewed papers.

Metric	Papers reporting (% of total)	Applications with the metric
HLS tool	49 (96%)	88 (95%)
Lines of code	12 (24%)	24 (26%)
Development time	8 (16%)	11 (12%)
Maximum clock frequency	36 (71%)	70 (75%)
Latency	15 (29%)	28 (30%)
Execution time	4 (8%)	7 (8%)
Data rate	19 (37%)	36 (39%)
FPGA LUTs	43 (84%)	79 (85%)
FPGA DFFs	38 (75%)	73 (78%)
FPGA slices (AMD)	6 (12%)	19 (20%)
FPGA DSPs	25 (49%)	48 (52%)
FPGA BRAMs	29 (57%)	49 (53%)

B. METRICS BREAKDOWN

Table 5 lists the investigated metrics from the surveyed papers and their frequency of occurrence. The table lists how many papers accepted to the survey reported on a given metric and how many applications the metric was mentioned for. The metrics are related to the HLS tools used, the development effort, the performance of the application, and the consumption of FPGA resources.

More than 90% of the articles listed the HLS tools used. This should be mandatory information in any HLS-related paper, but the omitting papers probably left the information out due to license reasons forbidding bench-marking. Development effort in terms of LoC or development time was listed in only 24% and 16% of the papers, respectively, which is disappointing as this information is crucial when comparing

TABLE 6. HLS vs. mRTL performance comparisons.

Metric	N	HLS/mRTL mean	Geom. std. dev.	HLS better or equal to mRTL
Max. clock freq.	70	0.97	1.65	53%
Latency	28	2.06	3.73	14%
Execution time	7	3.36	4.04	14%
Data rate	36	0.72	3.67	56%
Performance	79	0.67	3.04	41%

the productivity of the different methods. 69% of the surveyed articles used a self-made mRTL reference for comparison, and at least in these cases, the development effort for both flows should have been known. However, most authors were only interested in comparing the QoR.

The middle rows of the table show performance metrics that can be categorized as maximum clock frequency, latency, execution time, and data rate, even if individual papers used slightly different names for these. The maximum clock frequency refers to the achievable clock frequency of the application when synthesized on an FPGA. Latency refers to the time taken for a sample to propagate through the data path of the system, expressed in clock cycles or seconds. Execution time means the running time of the application for one complete set of inputs, and the data rate expresses how many discrete outputs the synthesized application could produce per second. Forty-two papers listed at least one of these, representing 82% of the total.

The final rows show FPGA resource consumption in look-up tables (LUTs), D flip-flops (DFFs), AMD FPGA slices, DSP blocks, and BRAMs. The main FPGA fabric consists of LUTs and DFFs programmed and connected to perform arithmetic and logical functions. Consumption of these elements was usually reported in the surveyed papers, but some papers listed the usage of slices instead. These are FPGA construction blocks specific to AMD FPGAs that contain a few LUTs and DFFs, the details depending on the exact FPGA model. Most FPGAs also contain pre-built DSP blocks dedicated to multiply-accumulate operations and on-chip random access memory elements (block RAMs, or BRAMs), which are also included in the table, as many applications utilize them.

C. QOR COMPARISONS

Table 6 aggregates the values related to the performance of the application from the survey data. Performance metrics are as reported in Table 5, with an additional row for combined performance, explained in the next paragraph. The second column gives the number of applications reported for the metric. The third column shows the geometric mean of the ratios of the HLS score vs. the mRTL score for a given metric, and the fourth column is the corresponding geometric standard deviation. Finally, the fifth column shows the percentage of applications for which the HLS score was better or equal to the mRTL score. Geometric mean was used rather than arithmetic average, as the data deals with ratios and because of the large variability between the data points.

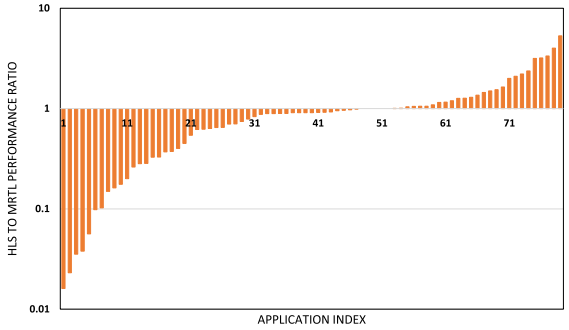

FIGURE 4. HLS to mRTL performance by application.

TABLE 7. HLS vs. mRTL Basic FPGA resource usage comparison.

Metric	N	HLS/mRTL mean	Geom. std. dev.	HLS better or equal to mRTL
LUTs	79	1.27	2.29	37%
DFFs	73	1.28	2.28	37%
Slices	19	1.63	1.39	5%
Basic FPGA res.	87	1.36	2.15	33%

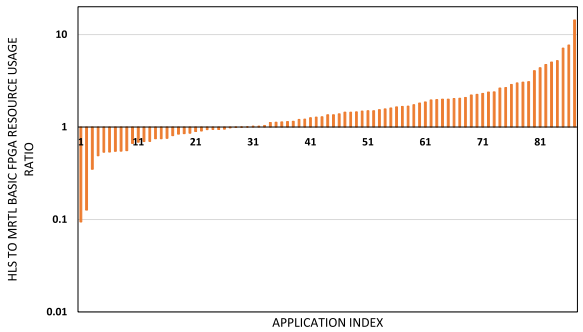

FIGURE 5. HLS to mRTL basic FPGA resource usage by application.

The geometric standard deviation should be interpreted so that most data points lie within the range from "mean divided by std. dev." to "mean multiplied by std. dev." For example, for the maximum clock frequency, most data lie between 0.59 and 1.60, with 0.97 being the mean.

The data show that HLS does not achieve as high performance as mRTL by geometric mean. This holds for all four performance metrics. Note that for latency and execution time, a ratio larger than one is favorable for mRTL, as one wants to minimize the values for those metrics. The additional metric, labeled *performance*, combines the data from all the other four metrics by selecting for each surveyed application the reported performance metric using the following priority: 1) data rate, 2) execution time, 3) latency, 4) max. clock frequency. In other words, if many performance metrics were reported for a single application, the one with the highest priority was used for the combined performance score. Reciprocals of latency and execution time ratios were used for the combined score to make a larger ratio favorable to HLS in all cases.

TABLE 8. HLS vs. mRTL DSP block and BRAM usage comparison.

	N	HLS ave.	mRTL ave.	Ratio	HLS better or equal to mRTL
DSP blocks	38	51.9	43.4	1.20	53 %
BRAMs (#)	38	100	103	0.97	61 %

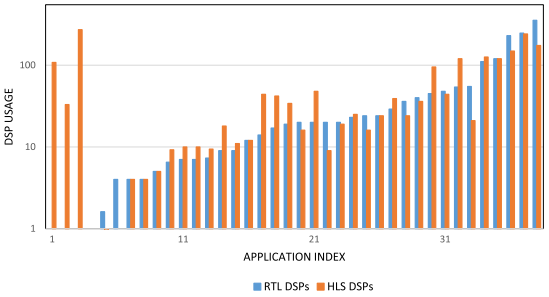

FIGURE 6. DSP usage by application.

The numbers show that in roughly 40% of the cases, the performance achieved was better with HLS than with mRTL, with the geometric mean of the ratios being 0.67. Another way of looking at the data is shown in Fig. 4, which depicts the combined performance ratio application by application, sorted from lowest to highest. An approximate observation can be made that in one third of the applications, the HLS performance was clearly worse than the mRTL performance; in one third, it was more or less equal and in one third, it was somewhat better.

Table 7 shows the relative usage of basic FPGA resources reported in the surveyed articles for each application. An aggregate metric *basic FPGA resources* was developed to combine the different values into one number. If Xilinx slices were reported, that value was used as a slice contains both LUTs and DFFs. Otherwise, the number of used LUTs and DFFs was summed to give the total consumption of basic FPGA resources. The table shows that HLS tends to use more of these basic resources than the same application implemented with mRTL, but the difference is slightly less pronounced than with performance metrics. A third of HLS applications fared better than mRTL in basic resource usage.

Fig. 5 shows the basic FPGA resource usage ratio application by application, sorted from lowest to highest. Note that application indices do not correspond to the same applications as in Fig. 4. Proportionally, a smaller number of applications have a large ratio favoring mRTL. Still, otherwise, the diagrams are quite similar after discounting that a large ratio now favors mRTL, whereas, in performance, it favors HLS.

Table 8 shows the DSP block and BRAM usage averages calculated for applications that reported using them. The geometric mean could not be used here because the data included zeros. Therefore, arithmetic averages of absolute values were used instead of calculating geometric means of ratios. Two outlier data points containing values on the order of magnitude larger than others were removed from the DSP

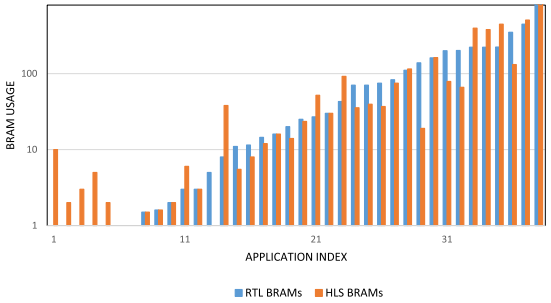

FIGURE 7. BRAM usage by application.

block and the BRAM average calculations to prevent them from skewing the results. For BRAMs, most papers reported usage in the number of BRAMs and some in kilobits. For simplicity, only the data for the number of BRAMs was used. DSP and BRAM usage data are further visualized in Figs. 6 and 7, which compare the DSP consumption and BRAM consumption application by application. The sorting of applications from left to right is in the order of increasing mRTL resource usage consumption. The few empty places contain data where both HLS and mRTL usage is 1.

It can be seen that the difference in DSP block and BRAM usage is not significant between the mRTL and HLS methods. The usage ratios are close to one, and even application-by-application the usage differences do not seem significant on average. Thus, it appears that although HLS seems to consume more basic FPGA resources than mRTL, there is no notable consumption difference in DSP blocks and BRAMs.

Performance and resource usage are typically trade-offs to each other. To increase performance, resource consumption usually needs to be increased by, e.g., adding parallelization. On the other hand, folding loops and increasing sharing reduce resource consumption at the cost of performance. Therefore, the two metrics should not only be compared separately but also combined for each application. Figs. 8 and 9 show the HLS to mRTL performance against the basic resource usage for each application for which both values were available. Here, performance and basic resources are defined as in tables 6 and 7. Note that DSP blocks and BRAMs are not included in the basic resource usage, even though many applications utilize them. Unfortunately, there is no feasible way to combine all the FPGA resources in a single metric, similar to the area on an ASIC. For example, the implementation of a DSP block varies between vendors and different FPGA families, and it is not trivial to convert a DSP block into an equivalent number of LUTs and DFFs. Nevertheless, the figure should give an indication of relative performance and resource consumption, especially since the difference in DSP and BRAM usage between HLS and mRTL was noticed to be not significant.

Fig. 8 shows the ratio of HLS to mRTL performance against the ratio of HLS to mRTL basic resource usage by application. Each "X" in the figure corresponds to one application. Break-even lines, where the ratios are 1, are emphasized. The area to the right and below the lines is

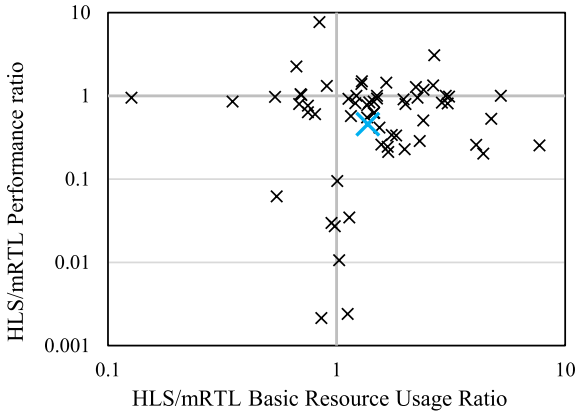

FIGURE 8. Relative HLS to mRTL performance to basic resource usage by application.

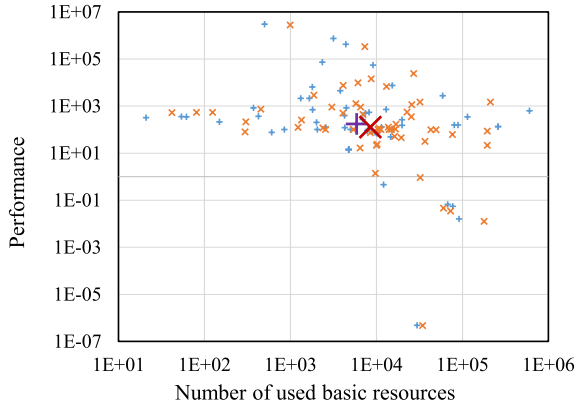

FIGURE 9. Absolute HLS and mRTL performance and basic resource usage by application.

where HLS has worse performance and higher resource consumption than mRTL, whereas the upper left corner favors HLS in both aspects. The large blue "X" indicates the center of gravity of the marks by geomean. It can be seen that the center of gravity favors mRTL in both performance and resource usage, but not by a significant margin.

Another way to look at the same data is depicted in Fig. 9, which shows the absolute performance and basic resource usage values for all HLS and mRTL applications separately. Each blue "+" indicates an mRTL application, and an orange "x" is an HLS application. The larger "+" and "x" signs show the centers of gravity of both clouds. The right way to look at this figure is not to concentrate on the values as such but on the shape and position of the two clouds relative to each other. It is easy to see that the clouds largely overlap, and the centers of gravity are not far apart. This confirms that mRTL outperforms and saves more basic FPGA resources than HLS in most cases, but only by a relatively small amount in both.

D. DEVELOPMENT EFFORT COMPARISONS

Table 9 shows the relative number of LoCs, development time, and development effort for HLS versus mRTL. Again, the geometric mean is used because the surveyed applications

TABLE 9. HLS vs. mRTL development effort comparisons.

Metric	N	HLS/mRTL mean	Geom. std. dev.	HLS better or equal to mRTL
Lines of code	24	0.47	1.91	88 %
Dev. time	11	0.24	1.72	100 %
Dev. effort	32	0.39	2.02	91 %

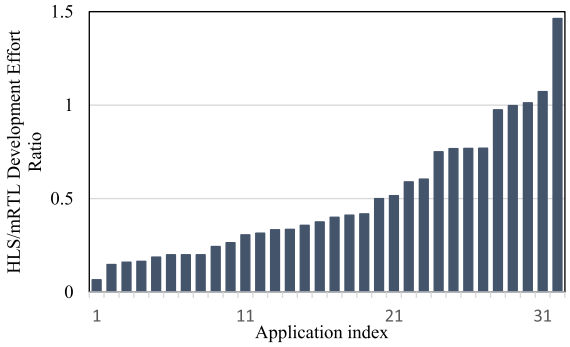

FIGURE 10. HLS to mRTL relative development effort by application.

varied in LoC from tens to more than ten thousand and in development time from a few hours to several months. The aggregate metric *development effort* was formed using development time if available and LoC otherwise. The values show that, on average, using HLS significantly saves development effort compared to mRTL. A saving of 50% to 75% can be expected.

Fig. 10 shows the development effort data, which depicts the HLS to mRTL development effort ratio application by application, ordered from lowest ratio to highest. In roughly a third of the cases, the development effort is 25% or less with HLS compared to mRTL, in another third, it is between 25% and 50%, and in the final third between 50% and 100%. Only one outlier case shows clearly higher development effort with HLS than with mRTL.

These figures verify the expectation that HLS should have a significant advantage over mRTL in the development effort. As an additional observation, a slight negative correlation was seen in the data for the relative development effort as a function of the absolute development effort. In other words, the larger the project, the less the HLS development effort tends to be compared to the mRTL development effort. Regarding LoC, the correlation coefficient was -0.24 and development time -0.35. This might be because in a small project the overhead of using HLS absorbs some of the benefits of employing it.

The development effort is tied to productivity, which can be defined as how much useful work is achieved per hour. "Useful work" does not have an established definition in the context of IC design. Still, one place-holder metric could be related to the performance of the finished and verified application. This should be compared with the development effort to get the productivity, for example, by dividing the performance by the development effort. The same productivity definition is used, for example, in [121] in

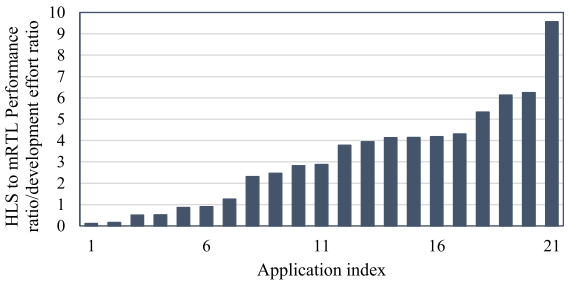

FIGURE 11. HLS to mRTL relative productivity by application.

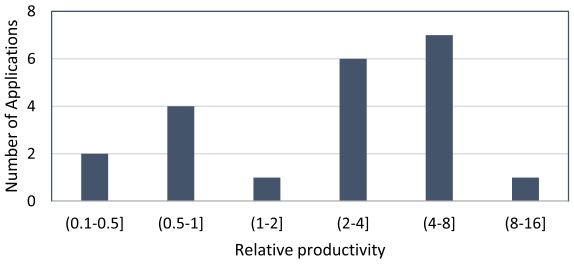

FIGURE 12. HLS to mRTL relative productivity histogram.

the software context. Figs. 11 and 12 show the productivity defined in this way for applications for which data on performance and development effort were available. Fig. 11 shows the relative HLS to mRTL performance divided by the relative development effort for 21 applications, while Fig. 12 shows a histogram of the same data. A large range of relative productivity from 0.1 to 9.6 can be observed, but in most cases higher productivity can be achieved with HLS. The variances could be caused by many factors, including the used HLS tool, the type of application, and the designers' experience, but this cannot be resolved from the available data.

E. OBSERVATIONS ON HLS TOOLS AND LANGUAGES

The data in the surveyed papers also present opportunities to see what HLS tools and languages are used today, at least in academia. Table 10 shows the breakdown of the applications by the HLS tool used to develop them. It is immediately apparent that Vitis HLS (formerly Vivado HLS) by AMD dominates here, with 69% of applications developed with it. The popularity of AMD FPGAs probably explains this, and the Vivado/Vitis HLS tool comes with the Vivado synthesis package. It is also based on C/C++ languages familiar to many hardware developers, making adoption straightforward. The other tools on the table receive more scattered usage, with only HDL Coder receiving more than five mentions. The table also shows that C-based languages dominate in HLS. Note that some tools in the table are no longer available (e.g., Altera OpenCL, LegUp).

Table 11 collects the language usage by application. 82% of the applications for which the data was disclosed were developed with either C or C++, confirming the high prevalence of these languages in HLS. OpenCL, SystemC, and SYCL are also based on C++ but have built-in features for parallelism. This leaves only 12 percent of the applications developed with non-C-based languages.

The popularity of Vivado/Vitis HLS allows for comparing results obtained with it to other HLS tools. Table 12 shows performance, basic FPGA resource usage, DSP block usage, and development effort geomeans against mRTL for Vivado/Vitis HLS compared against other tools. There does not appear to be difference in performance, but Vitis seems to infer more DSP blocks to save in basic FPGA resources than other HLS tools. On the other hand, this may be a property of the Vivado FPGA synthesis tool. The lower development effort might be because Vivado/Vitis uses the familiar C++ language, unlike many other tools, lowering its learning curve.

It would also be interesting to compare the QoR and development efforts of the other HLS tools against each other, but their limited usage makes this impossible with the survey data.

TABLE 10. HLS tool breakdown in reviewed papers.

HLS Tool	N	Tool languages
Altera/Intel OpenCL	2	OpenCL
Bambu	1	C/C++
Cadence Stratus	1	C/C++/SystemC
Intel HLS Compiler	3	C++
Kanagawa HLS	3	Kanagawa
LegUp	2	C/C++
MATLAB HDL Coder	6	MATLAB
Maxeler MaxCompiler	1	Java
Intel oneAPI	1	C++/SYCL
Siemens Catapult	3	C/C++/SystemC
Xilinx Vivado/Vitis	64	C/C++
XLS	1	DSLX
Undisclosed	5	N/A

TABLE 11. HLS language breakdown in reviewed papers.

Language	N
C/C++	73
DSLX	1
Java	1
Kanagawa	3
MATLAB	6
OpenCL	3
SYCL	1
SystemC	1
Undisclosed	4

TABLE 12. Comparison of Vivado/Vitis HLS to other HLS tools.

Metric	Vivado/Vitis	Others
Perf. geomean	0.67	0.67
Basic res. geomean	1.22	1.77
DSP blocks geomean	1.14	0.91
Dev. effort	0.35	0.44

VII. DISCUSSION

A. QOR AND PRODUCTIVITY OF HLS

Based on 93 applications surveyed, it was found that in 59% of the cases mRTL achieved higher performance than an HLS implementation of the same application. The geometric mean of the relative HLS to the mRTL performance was 0.67. Thus, it seems that higher performance can usually be reached with mRTL methods. Similarly, it was found that in 67% of the cases, an application implemented with mRTL methods consumed fewer basic FPGA resources than an HLS version of the application, with the mean relative consumption of HLS to mRTL being 1.36. Smaller differences were seen between the DSP block and BRAM consumptions, and there was no clear dominance for either method. Figs. 8 and 9 showed these data application by application, where the observation was that while mRTL favorably dominates HLS in both performance and the use of basic FPGA resources, the difference is relatively small. Power consumption was not compared due to the low amount of data, although it is also a significant quality metric.

The reason for the difference between the QoR of the HLS and mRTL methods is not apparent in this research. The two leading explanations are probably the optimization abilities of the HLS tools and the users' experience. Finding a high-quality micro-architecture from an abstract source code is a non-trivial task an HLS tool must perform. In addition, there is a gap between the higher-level model of the HLS tool and the considerations required to place and route a design on an FPGA [9]. On the other hand, it is commonly reported that HLS tools cannot produce a good QoR from an algorithm written with a software mindset, and many optimizations related to data widths, memory architecture, IO behavior, and concurrency are required [6], [122]. In line with this, a Siemens field application engineer with extensive HLS background reported in an interview that, with solid support, customers using Siemens Catapult HLS achieve at least as high QoR as with mRTL. However, without experienced engineers and vendor support, the results can be poor [123].

It would be invaluable if the papers surveyed in the previous section had information on the level of HLS expertise of those implementing the applications. This would allow HLS-mRTL QoR differences to be analyzed based on different experience levels, and perhaps this would emerge as the most significant factor explaining the differences in the results. Nevertheless, it is a limitation in HLS that high expertise is needed.

HLS is seen as a faster way to develop FPGA applications than mRTL. 88% of the projects had fewer LoC with HLS than with mRTL for the same application, with the relative LoC mean being 0.47 in favor of HLS. Likewise, in all 100% cases, the development time of the HLS project was lower than the development time of the mRTL, and the mean relative development time was 0.24. In more than half of the cases, the overall HLS development effort was half or less of the mRTL development effort. The Siemens field application engineer interview confirmed that with experienced engineers, a project time reduction of roughly 50% can be expected with HLS [123]. The lower development effort can be attributed to a higher abstraction level in design entry, eliminating the need to code for details related to clock, microarchitecture, and interfaces [3]. In addition, HLS enables faster verification due to less room for bugs to live at a higher abstraction level, by using software verification methods, higher simulation speed, and left shifting of the overall verification process.

In Subsection VI-D, a productivity metric was used that combined QoR in the performance achieved and the development effort. In roughly 75% of the cases an equal or higher productivity was achieved with HLS than with mRTL. It therefore seems that even though it appears to be more difficult to achieve high performance with HLS, the lower development effort tends to more than offset this.

TABLE 13. Comparison between 2019 and 2024 study results.

Metric	HLS/RTL mean 2019	HLS/RTL mean 2024	2024/2019 ratio
Lines of code	0.52 (36)	0.47 (24)	0.90
Development time	0.32 (25)	0.24 (11)	0.75
Productivity	4.4 (15)	3.2 (21)	0.73
Max. clock freq.	0.88 (74)	0.97 (70)	1.10
Latency	1.05 (17)	2.06 (28)	1.96
Execution time	1.70 (14)	3.36 (7)	1.98
Data rate	0.47 (46)	0.72 (36)	1.53
Basic FPGA res.	1.41 (92)	1.36 (87)	0.96
DSP Blocks	1.11 (50)	1.20 (38)	1.08
BRAMs	0.49 (29)	0.97 (38)	1.98

B. COMPARISON TO EARLIER STUDY

Table 13 compares the results of the previous [16] HLS survey to this one. The 2019 survey was based on years 2010-2016, while the current one is based on years 2017-2024. The metrics compared are shown as well as the HLS to RTL means for each metric. The numbers in parentheses are the N for the metric for each study. The final column shows the ratio of the means between the studies, with green indicating better numbers for the new study and red worse.

It can be seen that both development effort metrics have changed favorably, but this does not show directly in productivity, which has gone slightly down between studies. This may be due to performance metrics that have not changed as favorably. While data rate and maximum clock frequency metrics favor the new study, latency and execution time favor the older one. No significant changes can be seen in the FPGA resource usage metrics, except for BRAM usage, where the older study showed remarkably good results for HLS. In summary, it seems that while the HLS development effort relative to mRTL seems to be decreasing, there has been no clear improvement in QoR in recent years. Therefore, the previous study recommendation still holds that mRTL methods should be considered when the highest QoR is required, but otherwise HLS is a competitive method thanks to its favorable productivity.

Compared to the previous study, Vivado/Vitis has gained more popularity. In the 2019 article, it was used in 34% of the cases, while now it was used in 69% of them. The consolidation of HLS tools can also be seen in that the previous study reported the use of 19 different HLS tools, while the current one has only 12.

C. RECOMMENDATIONS

Fig. 1 showed the workflow used by most current HLS tools. The two most time-consuming steps in the flow are the ''modification for HLS'' and the ''design space exploration,'' as the other steps are already largely automated or related to the behavioral model, which must be made regardless of whether HLS or mRTL flow is used. Therefore, HLS tool developers should concentrate on these two steps to further reduce the HLS development effort and also to increase the QoR. Ideally, both steps should be largely automated, whereas they are now done manually.

In more detail, at least the following actions related to ''modification for HLS'' should be automated as far as possible:

1) Removal of dynamic memory handling and recursion. Querying the user for static limits could replace these with static equivalents. Pointers and system calls should also be removed or replaced with input from the user.
2) Replacing heavy mathematical operations such as division, sqrt, and trigonometric functions with hardware-friendly versions from pre-made libraries.
3) Likewise, replacing common functions from hardware-friendly libraries, such as those for sorting, searching, and linear algebra.
4) Inlining or mapping to the hierarchy of function calls according to optimization goals.
5) Optimizing parameter and variable bit-widths based on automated value range analysis.
6) Code refactoring to expose parallelism.

The design space exploration step involves inserting code pragmas or otherwise tuning the design for 1) loop pipelining and unrolling, 2) assigning arrays to registers or memories, 3) widening, splitting, and interleaving of memories, and 4) mapping IO to interface resources. This is a multidimensional optimization problem with a vast space to explore. Still, automation is possible, and heuristic algorithms concentrating on intermediate Pareto-optimal designs should be able to find adequate solutions. Approximate computing could also be used to prune the design space if a slight degradation in numerical accuracy is acceptable [124].

Even with the suggested automation, some actions will probably have to be performed manually. For example, the coding for clock-accurate design ([44]) and inserting feedback pipeline registers ([125]) seem operations that cannot be easily automated due to requiring an understanding of the behavioral properties of the application.

As seen in the previous section, most HLS tools use C++ as their primary input language. This is positive, as pragma-annotated C++ is well suited for HLS and has advantages in portability and popularity. It has also been shown in a previous article that development effort can be reduced, and thus productivity increased by adopting modern C++ as opposed to the old, more C-like 98/03 standards, which most HLS tools suggest to use in their tutorials and examples [69]. The study showed that at least two major C++-based HLS tools already have broad support for at least C++17. HLS tool vendors are encouraged to solidify support for modern language features and adopt them quickly as standards progress. In addition, hardware-friendly implementations of the Standard Template Library constructs should be provided to make source code translation from software model to hardware description more straightforward.

Finally, efforts should be made to add standardization to HLS flows. Even though most tools use C++ as the common input language for HLS, the different tools accept slightly different subsets of the language and can infer different hardware structures from the same source. Furthermore, tool-specific pragmas and libraries make porting HLS code from one tool to another laborious. It would accelerate the adoption of HLS if tool vendors developed and accepted an industry-wide standard of C++ for HLS. The standard would determine the synthesizable subset of C++ and define simulation and synthesis guidelines for different code constructs. A standard set of most widely needed pragmas should also be defined. This would allow HLS tool vendors to compete with quality of results, usability, and additional features while avoiding vendor locking customers.

VIII. CONCLUSION

This work studied the status of modern high-level synthesis for FPGAs from several perspectives. A survey of current HLS tools found 12 existing tools that take abstract, timeless, high-level code as input and produce RTL output based on technology and architectural constraints. Three of the tools were academic and the rest were commercial. It was noted that the number of HLS tools with ongoing support is slowly decreasing. C++-based HLS tools were found to be the majority. Platform support is diverse, with some tools targeting only vendor-specific FPGAs, some being vendor independent, and some being able to target ASICs.

A survey of published scientific articles from 2017 to 2024 comparing FPGA applications implemented with HLS and manual RTL (mRTL) methods examined HLS's QoR and productivity. The survey revealed that, on average, mRTL acquires somewhat better performance. HLS also consumed slightly more basic FPGA resources. However, a clearly lower development effort was observed with HLS and somewhat better overall productivity as well, calculated as the performance attained divided by the development effort.

The conclusion of the QoR/productivity survey is that, when a fast development process is needed, HLS is a competitive method. In contrast, mRTL should be considered when high performance and low resource consumption are

paramount. To achieve a QoR equivalent to that achievable by mRTL methods requires expertise and effort, reducing the productivity benefits of HLS. This emphasizes the need for HLS-trained engineers who understand the principles of hardware design but are also familiar with the input languages initially from the software domain. It seems unrealistic to expect that someone with a pure software background can use HLS as a jumping-pad to hardware design and expect a high QoR. HLS-related courses are therefore recommended for university computer engineering curricula.

Several suggestions were made to increase HLS productivity and QoR. First, HLS tools should be augmented with automated code refactoring tools to transform abstract software models into hardware-aware HLS input. Second, to make design space exploration faster, the tools should include automated exploration features to find solutions on the Pareto frontier. Third, the HLS tools should fearlessly adapt modern features from the software languages they are based on to reap the benefits that come along. Hardware-friendly implementations of commonly used software libraries should also be made available. Finally, standardization is suggested to draft a commonly accepted subset of C++ for HLS-based simulation and synthesis to enable better source code portability and production of vendor-independent utility tools.

The survey left open the question of what causes the variability in the QoR of HLS versus mRTL. The reason might lie in the differences between the HLS tools, the engineer's experience, or the type of application. A factor analysis is suggested as future research to find the underlying reasons. The survey also revealed that there is little research on the suitability of HLS for ASIC-based design, where awareness of the physical layout becomes more critical. Further research on this area is needed. Finally, while it was not a focus of this study, more research seems to be required on the effectiveness of verification in HLS. Although cosimulation between the abstract source and generated RTL is a standard feature of HLS tools, RTL output also requires independent verification. The availability of formal equivalence checking between the input and output of HLS would accelerate the overall verification process.

REFERENCES

[1] M. S. Lundstrom and M. A. Alam, "Moore's law: The journey ahead," *Science*, vol. 378, no. 6621, pp. 722–723, Nov. 2022.

[2] A. Takach, "High-level synthesis: Status, trends, and future directions," *IEEE Design Test*, vol. 33, no. 3, pp. 116–124, Jun. 2016.

[3] H. Ren, "A brief introduction on contemporary high-level synthesis," in *Proc. IEEE Int. Conf. IC Design Technol.*, May 2014, pp. 1–4.

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[5] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 8–17, Jul. 2009.

[6] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: Successes, challenges, and opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, pp. 1–42, Aug. 2022.

[7] K. Campbell, W. Zuo, and D. Chen, "New advances of high-level synthesis for efficient and reliable hardware design," *Integration*, vol. 58, pp. 189–214, Jun. 2017.

[8] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, "Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains," *IEEE Access*, vol. 8, pp. 174692–174722, 2020.

[9] Y.-H. Lai, E. Ustun, S. Xiang, Z. Fang, H. Rong, and Z. Zhang, "Programming and synthesis for software-defined FPGA acceleration: Status and future prospects," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 14, no. 4, pp. 1–39, Sep. 2021.

[10] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 18–25, Jul. 2009.

[11] J. Matai, D. Richmond, D. Lee, and R. Kastner, "Enabling FPGAs for the masses," 2014, *arXiv:1408.5870*.

[12] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *Proc. 9th IEEE Int. Conf. ASIC*, Oct. 2011, pp. 1102–1105.

[13] Z. Sun, K. Campbell, W. Zuo, K. Rupnow, S. Gurumani, F. Doucet, and D. Chen, "Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis," in *Proc. 21st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2016, pp. 218–225.

[14] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1014–1029, May 2021.

[15] E. D. Sozzo, D. Conficconi, A. Zeni, M. Salaris, D. Sciuto, and M. D. Santambrogio, "Pushing the level of abstraction of digital system design: A survey on how to program FPGAs," *ACM Comput. Surveys*, vol. 55, no. 5, pp. 1–48, Dec. 2022.

[16] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019.

[17] R. S. Molina, V. Gil-Costa, M. L. Crespo, and G. Ramponi, "High-level synthesis hardware design for FPGA-based accelerators: Models, methodologies, and frameworks," *IEEE Access*, vol. 10, pp. 90429–90455, 2022.

[18] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, Oct. 2020.

[19] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Autom. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, Sep. 2012.

[20] R. Nane, V.-M. 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[21] S. Ravi and M. Joseph, "Open source HLS tools: A stepping stone for modern electronic CAD," in *Proc. IEEE Int. Conf. Comput. Intell. Comput. Res. (ICCIC)*, Dec. 2016, pp. 1–8.

[22] Clash-lang.org. (2023). *Clash*. [Online]. Available: https://clash-lang.org/

[23] ChipsAlliance. (2024). *Chisel Software-defined Hardware*. [Online]. Available: https://www.chisel-lang.org/

[24] S. Sato and K. Kise, "ArchHDL: A new hardware description language for high-speed architectural evaluation," in *Proc. IEEE 7th Int. Symp. Embedded Multicore Socs*, Sep. 2013, pp. 107–112.

[25] S. Takamaeda-Yamazaki, "Pyverilog: A Python-based hardware design processing toolkit for verilog HDL," in *Applied Reconfigurable Computing*. Cham, Switzerland: Springer, 2015, pp. 451–460.

[26] R. Nikhil, "Bluespec system verilog: Efficient, correct RTL from high level specifications," in *Proc. 2nd ACM IEEE Int. Conf. Formal Methods Models Co-Design*, Jun. 2004, pp. 69–70.

[27] Redwood EDA. (2024). *Tl-verilog*. [Online]. Available: https://www.redwoodeda.com/tl-verilog

[28] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "HIPAcc: A domain-specific language and compiler for image processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 210–224, Jan. 2016.

[29] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," 2016, *arXiv:1610.09405*.

[30] Y.-H. Lai, H. Rong, S. Zheng, W. Zhang, X. Cui, Y. Jia, J. Wang, B. Sullivan, Z. Zhang, Y. Liang, Y. Zhang, J. Cong, N. George, J. Alvarez, C. Hughes, and P. Dubey, "SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2020, pp. 1–9.

[31] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 296–311, Jun. 2018.

[32] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, New York, NY, USA, Feb. 2019, pp. 242–251.

[33] M. A. Özkan, A. Pérard-Gayot, R. Membarth, P. Slusallek, R. Leißa, S. Hack, J. Teich, and F. Hannig, "AnyHLS: High-level synthesis with partial evaluation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3202–3214,v Nov. 2020.

[34] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed., Reading, MA, USA: Addison-Wesley, 2006.

[35] Siemens Digit. Industries Softw. (2024). *Catapult High-Level Synthesis and Verification*. [Online]. Available: https://eda.sw.siemens.com/en-U.S./ic/catapult-high-level-synthesis/

[36] Z. Wang and B. C. Schafer, "Learning from the past: Efficient high-level synthesis design space exploration for FPGAs," *ACM Trans. Design Autom. Electron. Syst.*, vol. 27, no. 4, pp. 1–23, Feb. 2022.

[37] LLVM Admin Team. (2024). *The LLVM Compiler Infrastructure*. [Online]. Available: https://llvm.org/

[38] S. A. Edwards, "The challenges of hardware synthesis from C-like languages," in *Proc. Design, Autom. Test Eur.*, Mar. 2005, pp. 66–67.

[39] Y. Herklotz, Z. Du, N. Ramanathan, and J. Wickerson, "An empirical study of the reliability of high-level synthesis tools," in *Proc. IEEE 29th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, May 2021, pp. 219–223.

[40] J. Hu, T. Li, and S. Li, "Equivalence checking between SLM and RTL using machine learning techniques," in *Proc. 17th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2016, pp. 129–134.

[41] J. Hu, Y. Hu, L. Yu, W. Wang, H. Yang, Y. Kang, and J. Cheng, "Formal verification of GCSE in the scheduling of high-level synthesis: Work-in-Progress," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Sep. 2020, pp. 1–2.

[42] R. Chouksey, C. Karfa, and P. Bhaduri, "Formal verification of optimizing transformations during high-level synthesis," in *Proc. 12th Innov. Softw. Eng. Conf.*, New York, NY, USA, Feb. 2019, pp. 1–5.

[43] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, "Formal verification of high-level synthesis," in *Proc. ACM Program. Lang.*, vol. 5, Oct. 2021, pp. 1–30.

[44] S. Lahti, J. Vanne, and T. D. Hämäläinen, "Designing a clock cycle accurate application with high-level synthesis," in *Proc. IECON-42nd Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2016, pp. 4756–4761.

[45] Lombiq Technol. (2024). *Hastlayer*. [Online]. Available: https://hastlayer.com/

[46] (2024). *Cacheq*. [Online]. Available: https://cacheq.com/

[47] Julian Kemmerer. (2024). *Pipelinec*. [Online]. Available: https://github.com/JulianKemmerer/PipelineC

[48] B. Pelton, A. Sapek, K. Eguro, D. Lo, A. Forin, M. Humphrey, J. Xi, D. Cox, R. Karandikar, J. de Fine Licht, E. Babin, A. Caulfield, and D. Burger, "Wavefront threading enables effective high-level synthesis," in *Proc. ACM Program. Lang.*, vol. 8, Jun. 2024, pp. 1066–1090.

[49] M. Levental, A. Khan, R. Chard, K. Yoshii, K. Chard, and I. Foster, "OpenHLS: High-level synthesis for low-latency deep neural networks for experimental science," 2023, *arXiv:2302.06751*.

[50] Universität des Saarlandes. (2024). *Hipacc*. [Online]. Available: https://hipacc-lang.org/

[51] S. Skalicky, J. Monson, A. Schmidt, and M. French, "Hot & spicy: Improving productivity with Python and HLS for FPGAs," in *Proc. IEEE 26th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 85–92.

[52] PandA Team. (2024). *Bambu*. [Online]. Available: https://panda.deib.polimi.it/?page_id=31

[53] OpenROAD Project. (2024). *Openroad*. [Online]. Available: https://theopenroadproject.org/

[54] NEC. (2024). *Cyberworkbench*. [Online]. Available: https://www.nec.com/en/global/prod/cwb/index.html?

[55] EPFL. (2024). *Dynamatic from C/c++ to Dynamically-Scheduled Circuits*. [Online]. Available: https://dynamatic.epfl.ch/

[56] MathWorks. (2024). *HDL Coder*. [Online]. Available: https://se.mathworks.com/products/hdl-coder.html

[57] Intel. (2024). *Intel High Level Synthesis Compiler*. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[58] Intel. (2024). *Pending Deprecation of the Intel HLS Compiler*. Accessed: Dec. 19, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683680/23-2/pending-deprecation-of-the.html

[59] Intel. (2024). *Intel Oneapi Dpc++/c++ Compiler*. Accessed: Dec. 19, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html

[60] Babar Khan Niazi. (2024). *Kiwi-Compiler-HLS*. Accessed: Dec. 19, 2024. [Online]. Available: https://github.com/BabarZKhan/Kiwi-compiler-HLS

[61] Khronos Group. (2024). *Sycl*. Accessed: Dec. 19, 2024. [Online]. Available: https://www.khronos.org/api/index_2017/sycl

[62] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, New York, NY, USA, Feb. 2011, pp. 33–36.

[63] Microchip. (2024). *Smarthls Compiler Software*. Accessed: Dec. 19, 2024. [Online]. Available: https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler

[64] Siemens. (2024). *Modelsim*. Accessed: Dec. 19, 2024. [Online]. Available: https://eda.sw.siemens.com/en-U.S./ic/modelsim/

[65] Cadence. (2024). *Stratus High-Level Synthesis*. Accessed: Dec. 19, 2024. [Online]. Available: https://www.cadence.com/en_US/home/resources/datasheets/stratus-high-level-synthesis-ds.html

[66] AMD. (2024). *Amd Vitis HLS*. Accessed: Dec. 19, 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

[67] AMD. (2024). *Amd Vitis Unified Software Platform*. Accessed: Dec. 19, 2024. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html

[68] XLS Community. (2024). *XLS: Accelerated HW Synthesis*. Accessed: Dec. 19, 2024. [Online]. Available: https://google.github.io/xls/

[69] S. Lahti, M. Rintala, and T. D. Hämäläinen, "Leveraging modern C++ in high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 4, pp. 1123–1132, Apr. 2023.

[70] B. Mohamed, A. Elsayed, O. Amin, E. Khafagy, M. Abdelrasoul, A. Shalaby, and M. S. Sayed, "High-level synthesis hardware implementation and verification of HEVC DCT on SoC-FPGA," in *Proc. 13th Int. Comput. Eng. Conf. (ICENCO)*, Dec. 2017, pp. 361–365.

[71] F. Winterstein, "High-level synthesis of dynamic data structures," in *Separation Logic for High-Level Synthesis*. Cham, Switzerland: Springer, 2017, pp. 11–33.

[72] M. A. Hussain, R. Badar, and S. W. Nabi, "Comparison of hand-written RTL code against high-level synthesis for blowfish and tiny encrpytion algorithm (TEA)," in *Proc. Int. Conf. FPGA Reconfig. General-Purpose Comput. (FPGA4GPC)*, Mar. 2017, pp. 1–6.

[73] A. Stanciu and C. Gerigan, "Comparison between implementations efficiency of HLS and HDL using operations over Galois fields," in *Proc. IEEE 23rd Int. Symp. Design Technol. Electron. Packag. (SIITME)*, Oct. 2017, pp. 171–174.

[74] P. Sjövall, V. Viitamäki, J. Vanne, and T. D. Hämäläinen, "High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2017, pp. 1547–1551.

[75] I. Stamoulias, C. Kachris, and D. Soudris, "Hardware accelerators for financial applications in HDL and high level synthesis," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Modeling, Simulation (SAMOS)*, Jul. 2017, pp. 278–285.

[76] N. Fujieda, S. Ichikawa, Y. Ishigaki, and T. Tanaka, "Evaluation of the hardwired sequence control system generated by high-level synthesis," in *Proc. IEEE 26th Int. Symp. Ind. Electron. (ISIE)*, Jun. 2017, pp. 1261–1267.

[77] Y. Afsharnejad, A.-A. Yassine, O. Ragheb, P. Chow, and V. Betz, "HLS-based FPGA acceleration of light propagation simulation in turbid media," in *Proc. 9th Int. Symp. Highly-Efficient Accel. Reconfigurable Technol.*, New York, NY, USA, Jun. 2018, pp. 1–6.

[78] Y. Barrios, A. Sanchez, L. Santos, S. López, J. Fco. López, and R. Sarmiento, "Hardware implementation of the CCSDS 123.0-B-1 lossless multispectral and hyperspectral image compression standard by means of high level synthesis tools," in *Proc. 9th Workshop Hyperspectral Image Signal Process., Evol. Remote Sens. (WHISPERS)*, Sep. 2018, pp. 1–5.

[79] N. Dimou, M. Lourakis, G. Lentaris, D. Soudris, and D. Reisis, "Parallel robust absolute orientation on FPGA for vision and robotics," in *Proc. 25th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2018, pp. 665–668.

[80] J. Marjanovic, "Low vs high level programming for FPGA," in *Proc. 7th Int. Beam Instrum. Conf.*, 2018, pp. 527–533.

[81] J. Santiago Da Silva, F.-R. Boyer, and J. M. P. Langlois, "P4-compatible high-level synthesis of low latency 100 Gb/s streaming packet parsers in FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, New York, NY, USA, Feb. 2018, pp. 147–152.

[82] T. Marc-André, "Two FPGA case studies comparing high level synthesis and manual HDL for HEP applications," 2018, *arXiv:1806.10672*.

[83] T. Hänninen, H. Y. Amin, and M. Juntti, "Base station MIMO detector algorithm implementations," in *Proc. 52nd Asilomar Conf. Signals, Syst., Comput.*, Oct. 2018, pp. 195–198.

[84] S. Bansal, H. Hsiao, T. Czajkowski, and J. H. Anderson, "High-level synthesis of software-customizable floating-point cores," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 37–42.

[85] B. Bilgili, C. Yamaneren, K. Vatansever, U. Çoltu, and B. Örs, "System on chip design with Vivado high-level synthesis tool," in *Proc. 11th Int. Conf. Electr. Electron. Eng. (ELECO)*, Nov. 2019, pp. 1047–1050.

[86] Y. Delomier, B. L. Gal, J. Crenne, and C. Jego, "Generation of efficient self-adaptive hardware polar decoders using high-level synthesis," in *Proc. IEEE Int. Workshop Signal Process. Syst. (SiPS)*, Oct. 2019, pp. 242–247.

[87] F. Farahmand, D. T. Nguyen, V. B. Dang, A. Ferozpuri, and K. Gaj, "Software/Hardware codesign of the post quantum cryptography algorithm NTRUEncrypt using high-level synthesis and register-transfer level design methodologies," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 225–231.

[88] T. Ohkawa, Y. Sugata, H. Watanabe, N. Ogura, K. Ootsu, and T. Yokota, "High level synthesis of ROS protocol interpretation and communication circuit for FPGA," in *Proc. IEEE/ACM 2nd Int. Workshop Robot. Softw. Eng. (RoSE)*, May 2019, pp. 33–36.

[89] A. Varici, G. Saglam, S. Ipek, A. Yildiz, S. Gören, A. Aysu, D. Iskender, T. B. Aktemur, and H. F. Ugurdag, "Fast and efficient implementation of lightweight crypto algorithm PRESENT on FPGA through processor instruction set extension," in *Proc. IEEE East-West Design Test Symp. (EWDTS)*, Sep. 2019, pp. 1–5.

[90] M. Moghaddamfar, C. Färber, W. Lehner, and N. May, "Comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA," in *Proc. 16th Int. Workshop Data Manage. New Hardw.*, New York, NY, USA, Jun. 2020, pp. 1–7.

[91] P. Sikka, A. R. Asati, and C. Shekhar, "Power- and area-optimized high-level synthesis implementation of a digital down converter for software-defined radio applications," *Circuits, Syst., Signal Process.*, vol. 40, no. 6, pp. 2883–2894, Jun. 2020.

[92] W. Altoyan and J. J. Alonso, "Investigating performance losses in high-level synthesis for stencil computations," in *Proc. IEEE 28th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2020, pp. 195–203.

[93] S. Luthra, M. A. S. Khalid, and M. A. Moin Oninda, "FPGA-based evaluation and implementation of an automotive RADAR signal processing system using high-level synthesis," in *Proc. IEEE Can. Conf. Electr. Comput. Eng. (CCECE)*, Aug. 2020, pp. 1–6.

[94] D. T. Nguyen, V. B. Dang, and K. Gaj, "High-level synthesis in implementing and benchmarking number theoretic transform in lattice-based post-quantum cryptography using software/hardware codesign," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, F. Rincón, J. Barba, H. K. H. So, P. Diniz, and J. Caba, Eds., Cham, Switzerland: Springer, 2020, pp. 247–257.

[95] H. S. Lee and J. W. Jeon, "Comparison between HLS and HDL image processing in FPGAs," in *Proc. IEEE Int. Conf. Consum. Electron.-Asia (ICCE-Asia)*, Nov. 2020, pp. 1–2.

[96] R. Millón, E. Frati, and E. Rucci, "A comparative study between HLS and HDL on SoC for image processing applications," *Elektron*, vol. 4, no. 2, pp. 100–106, Dec. 2020.

[97] S. Puranik, M. Barve, D. Shah, S. Sinha, R. Patrikar, and S. Rodi, "Key-value store using high level synthesis flow for securities trading system," in *Proc. Int. Conf. Comput., Electron. Commun. Eng. (iCCECE)*, Aug. 2020, pp. 237–242.

[98] M. A. Mansoori and M. R. Casu, "High level design of a flexible PCA hardware accelerator using a new block-streaming method," *Electronics*, vol. 9, no. 3, p. 449, Mar. 2020.

[99] P. Mantovani, R. Margelli, D. Giri, and L. P. Carloni, "HL5: A 32-bit RISC-V processor designed with high-level synthesis," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Mar. 2020, pp. 1–8.

[100] A. M. Salman, A. S. Tulan, R. Y. Mohamed, M. H. Zakhari, and H. Mostafa, "Comparative study of hardware accelerated convolution neural network on PYNQ board," in *Proc. 2nd Novel Intell. Lead. Emerg. Sci. Conf. (NILES)*, Oct. 2020, pp. 578–582.

[101] A. C. Mert, E. Karabulut, E. Öztürk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Trans. Comput.*, vol. 71, no. 11, pp. 2829–2843, Nov. 2020.

[102] P. Sikka, A. R. Asati, and C. Shekhar, "Real time FPGA implementation of a high speed and area optimized Harris corner detection algorithm," *Microprocessors Microsystems*, vol. 80, Feb. 2021, Art. no. 103514.

[103] P. Socha, V. Miškovský, and M. Novotný, "High-level synthesis, cryptography, and side-channel countermeasures: A comprehensive evaluation," *Microprocessors Microsystems*, vol. 85, Sep. 2021, Art. no. 104311.

[104] P. Sjövall, M. Rasinen, A. Lemmetti, and J. Vanne, "High-level synthesis implementation of an accurate HEVC interpolation filter on an FPGA," in *Proc. IEEE Nordic Circuits Syst. Conf. (NorCAS)*, Oct. 2021, pp. 1–7.

[105] D. Soni and R. Karri, "Efficient hardware implementation of PQC primitives and PQC algorithms using high-level synthesis," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2021, pp. 296–301.

[106] T. Saikai, K. Miyata, T. Manabe, and Y. Shibata, "Evaluation of an HLS-based heterogeneous redundant design approach for functional safety systems on FPGAs," in *Proc. 9th Int. Symp. Comput. Netw. (CANDAR)*, Nov. 2021, pp. 162–167.

[107] E. Zamiri, A. Sanchez, M. Yushkova, M. S. Martínez-García, and A. de Castro, "Comparison of different design alternatives for hardware-in-the-loop of power converters," *Electronics*, vol. 10, no. 8, p. 926, Apr. 2021.

[108] I. Hamzaoglu, H. Mahdavi, and E. Taskin, "FPGA implementations of VVC fractional interpolation using high-level synthesis," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2022, pp. 1–6.

[109] W. Li, N. Bartzoudis, J. R. Fernández, D. López-Bueno, G. Montoro, and P. Gilabert, "Slow-envelope shaping function FPGA implementation for 5G NR envelope tracking PA," in *Proc. Int. Workshop Integr. Nonlinear Microw. Millimetre-Wave Circuits (INMMiC)*, Apr. 2022, pp. 1–3.

[110] M. Vaziri and H. Jahanirad, "Highly efficient implementation of chaotic systems utilizing high-level synthesis tools," in *Proc. 30th Int. Conf. Electr. Eng. (ICEE)*, May 2022, pp. 501–506.

[111] S. A. Alam, D. Gregg, G. Gambardella, T. Preusser, and M. Blott, "On the rtl implementation of finn matrix vector compute unit," 2022, *arXiv:2201.11409*.

[112] G. Elsayed and S. Kayed, "A comparative study between MATLAB HDL coder and VHDL for FPGAs design and implementation," *J. Int. Soc. Sci. Eng.*, vol. 4, no. 4, pp. 92–98, Sep. 2022.

[113] A. Gondhalekar, T. Twomey, and W.-C. Feng, "On the characterization of the performance-productivity gap for FPGA," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2022, pp. 1–8.

[114] X. Wei and X. Guo, "Beyond verilog: Evaluating chisel versus high-level synthesis with tiny designs," in *Proc. 23rd Int. Symp. Quality Electron. Design (ISQED)*, Apr. 2022, p. 1.

[115] T. Havinga, X. Jiao, W. Liu, and I. Moerman, "Accelerating FPGA-based Wi-Fi transceiver design and prototyping by high-level synthesis," in *Proc. IEEE 31st Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2023, p. 219.

[116] T. Janson and U. Kebschull, "Data pre-processing with high-level-synthesis and dataflow programming using HLS C++ dataflow template library," *Nucl. Instrum. Methods Phys. Res. A, Accel. Spectrom. Detect. Assoc. Equip.*, vol. 1045, Jan. 2023, Art. no. 167594.
[117] A. Kamkin, M. Chupilko, M. Lebedev, S. Smolov, and G. Gaydadjiev, "High-level synthesis versus hardware construction," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Apr. 2023, pp. 1–6.
[118] P. Sikka, "High-level synthesis assisted, low-latency, area- and power-optimized FPGA implementation of MUSIC algorithm for direction-of-arrival estimation," *Sustain. Energy Technol. Assessments*, vol. 57, Jun. 2023, Art. no. 103201.
[119] S. Puranik, M. Barve, S. Rodi, and R. Patrikar, "Acceleration of trading system back end with FPGAs using high-level synthesis flow," *Electronics*, vol. 12, no. 3, p. 520, Jan. 2023.
[120] Bluespec Inc. (2024). *Bluespec Compiler*. Accessed: Dec. 19, 2024. [Online]. Available: https://github.com/B-Lang-org/bsc
[121] A. Funk, V. Basili, L. Hochstein, and J. Kepner, "Application of a development time productivity metric to parallel software development," in *Proc. 2nd Int. Workshop Softw. Eng. High Perform. Comput. Syst. Appl.*, New York, NY, USA, May 2005, pp. 8–12.
[122] M. Fingeroff, *High-Level Synthesis Blue Book*. Bloomington, IN, USA: AuthorHouse, 2010.
[123] P. Solanti, "Interview on the state of high-level synthesis," Siemens, Munich, Germany, May 2024.
[124] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb. 2016.
[125] S. Lahti, P. P. Campo, V. Lampu, L. Anttila, M. Valkama, and T. D. Hämäläinen, "Implementation of a nonlinear self-interference canceller using high-level synthesis," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

SAKARI LAHTI received the M.Sc. (Tech.) degree in engineering physics and the M.Sc.(Tech.) degree in computer engineering from Tampere University of Technology, Tampere, Finland, in 2002 and 2014, respectively.

From 2014 to 2017, he was a Doctoral Researcher with Tampere University of Technology. He is currently a University Instructor with the Unit of Computing Sciences, Tampere University, Tampere. His current research interests include high-level synthesis of digital systems and hardware and system-on-chip designing, in general.

TIMO D. HÄMÄLÄINEN (Member, IEEE) received the M.Sc. (Tech.) and Ph.D. degrees in electrical engineering from Tampere University of Technology, Tampere, Finland, in 1993 and 1997, respectively.

He is currently a Full Professor and the Head of the Unit of Computing Sciences, Tampere University, Tampere. He is the author of more than 90 journal articles and 250 conference publications. He holds several patents. His current research and teaching activities include system-on-chip design methodologies and tools, as well as SoC architectures and systems.