Are We There Yet? A Study on the State of High-Level Synthesis

Sakari Lahti, *Graduate Student Member, IEEE*, Panu Sjövall, *Graduate Student Member, IEEE*, Jarno Vanne, *Member, IEEE*, and Timo D. Hämäläinen, *Member, IEEE*

***Abstract*—To increase productivity in designing digital hardware components, high-level synthesis (HLS) is seen as the next step in raising the design abstraction level. However, the quality of results (QoRs) of HLS tools has tended to be behind those of manual register-transfer level (RTL) flows. In this paper, we survey the scientific literature published since 2010 about the QoR and productivity differences between the HLS and RTL design flows. Altogether, our survey spans 46 papers and 118 associated applications. Our results show that on average, the QoR of RTL flow is still better than that of the state-of-the-art HLS tools. However, the average development time with HLS tools is only a third of that of the RTL flow, and a designer obtains over four times as high productivity with HLS. Based on our findings, we also present a model case study to sum up the best practices in comparative studies between HLS and RTL. The outcome of our case study is also in line with the survey results, as using an HLS tool is seen to increase the productivity by a factor of six. In addition, to help close the QoR gap, we present a survey of literature focused on improving HLS. Our results let us conclude that HLS is currently a viable option for fast prototyping and for designs with short time to market.**

***Index Terms*—Electronic design automation and methodology, field-programmable gate array (FPGA), hardware description languages (HDLs), high-level synthesis (HLS), reconfigurable logic.**

I. Introduction

FOR DECADES now, register-transfer level (RTL) has been the dominant method to describe very large scale integration (VLSI) systems and their constituent intellectual property blocks. Whereas the RTL tools have developed only incrementally, the complexity of the VLSI systems has raised exponentially, which has made the design and verification process a bottleneck for productivity [1].

High-level synthesis (HLS) promises to alleviate this problem by a variety of ways [2]–[5]. In HLS, the application is described on a behavioral level, omitting implementation details such as timing and the nature of interface and memory elements. These details are determined using an HLS tool that takes the behavioral description as an input. The designer can select the target technology in the tool and map the interface and memory variables to specified technology-dependent elements. The HLS tool then produces an RTL description based on the target technology and microarchitectural choices.

The promises of HLS are many.

1) Initial design effort is reduced by raising the abstraction level. The designer can concentrate on describing the behavior of the system without having to spend time implementing the microarchitectural details. Introduction of bugs in the code is also less likely on a higher level of abstraction.
2) Verification is accelerated. The behavior of the design can often be verified using software verification tools that are faster and simpler to use than RTL simulation tools. Furthermore, the RTL output of the HLS tool can be verified by using the original behavioral test bench, as the tool can check that the results of both models are identical.
3) Design space exploration (DSE) is faster. The microarchitecture can be explored by making choices in the HLS tool, which require little or no modifications to the code. Thus, several transformations such as pipelining and various loop unrolling factors can be explored in a matter of hours. This is a tremendous improvement upon RTL methodology, where these kind of changes would require significant modifications to the source code.
4) Targeting new platforms is straightforward. If the target platform changes, the HLS tool is able to modify the RTL output accordingly. For example, if the new platform has a different clock frequency, the HLS tool reschedules operations according to the new frequency.
5) HLS is accessible to software engineers. Whereas RTL design requires knowledge of languages such as VHDL and Verilog, HLS tools usually use familiar languages such as C/C++. The HLS tool can take care of most of the hardware specific implementation details, so the threshold of software engineers to tackle hardware projects is greatly reduced. That said, to obtain optimal results, hardware expertise is still useful when employing HLS.

Together, these benefits reduce the design and verification time, push down the development costs, and lower the bar for undertake hardware projects. Consequently, the time

Manuscript received September 7, 2017; revised December 18, 2017 and March 12, 2018; accepted April 25, 2018. Date of publication May 8, 2018; date of current version April 19, 2019. This work was supported in part by the European Celtic-Plus Project under Grant 4KREPROSYS and in part by the Academy of Finland under Grant 301820. This paper was recommended by Associate Editor J. Cortadella. *(Corresponding author: Sakari Lahti.)*

The authors are with the Laboratory of Pervasive Computing, Tampere University of Technology, 33101 Tampere, Finland (e-mail: sakari.lahti@tut.fi).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2834439

to market is shortened, and using hardware acceleration on heterogeneous systems becomes a more attractive option.

The rise of field-programmable gate arrays (FPGAs) is also an enabling factor for HLS. FPGAs are ideal platforms for HLS designs, as they allow quick prototyping, have rapid design cycle, and are inherently reprogrammable. Modern HLS tools usually contain a wide library of FPGA technologies for design targeting.

The history of HLS dates back to the 1970s and 1980s, but it was not until the turn of the century that it became a viable option for the industry [2]. One of the reasons for the slow adoption is that the quality of results (QoR), such as resource usage and performance, was initially poor compared with the RTL approach. The QoR has improved with the newest generation of HLS tools, but the results reported in individual studies still vary, and it is unclear whether the QoR gap has been closed yet.

The goal of this paper is to answer this question by a literature review. We examine 46 recent papers that compare the QoR and development effort of HLS and RTL approaches for the same applications. This paper has four main contributions.

1) A comparative analysis of the QoR and design effort of HLS and RTL flows reported in scientific articles.
2) A case study presenting the best practices for comparing HLS and RTL approaches with a test group that uses both flows to implement a part of an HEVC/H.265 video encoder.
3) A survey of the literature suggesting research directions and ways to improve HLS.
4) The conclusions on the current state-of-the-art in HLS.

To the best of our knowledge, this is the first comprehensive quantitative study that uses a wide variety of sources to compare the QoR and design effort of HLS and RTL flows. Previous works have instead focused on comparing different HLS tools to each other [6], [7]. Other papers have provided insights on how to close the QoR gap against RTL or otherwise improve HLS tools [5], [8]. A thorough quantitative analysis on the current state of HLS has been missing, however, which this paper amends.

The rest of this paper is structured as follows. Section II describes our criteria for selecting papers for the quantitative analysis. Section III contains a meta-analysis of the reviewed papers, summarizing what kind of information was reported in them. In Section IV, we show and analyze the results from the literature study, and Section V describes our test group study with its results. Section VI reviews papers that propose improvements to HLS, and finally Section VII concludes this paper with some discussion of the results.

II. Qualifying Papers

For this paper, we examined articles published in 2010 or later to get a comprehensive view of the latest HLS works. Altogether, we found over a thousand candidate papers and selected those articles for further study whose abstracts stated that: 1) one or more applications were implemented using HLS and 2) the obtained results were compared with equivalent self-made or referenced RTL applications.

TABLE I
Number of Papers Published by Year

Year	Papers
2010	4
2011	5
2012	3
2013	8
2014	10
2015	8
2016	8

We also required the qualifying papers to list one or more of the following metrics for *both* the HLS and RTL versions of the applications.

1) Performance with an application specific metric.
2) Execution time and/or latency.
3) Maximum achievable clock frequency on target platform.
4) Area on application-specific integrated circuit (ASIC).
5) Resource usage on FPGA.
6) Power consumption.
7) Development time.
8) Lines of input source code (LoC).

In total, we found 46 qualifying papers out of which 39 were from IEEE Xplore, two from Springer Link, one from ACM Digital Library, two from arXiv.org, one from EBSCOhost, and one from Science Direct. Basic information on all the reviewed papers is given in Table IX in the Appendix. As can be seen from the table, the range of applications is very diverse. This makes it impractical to analyze the QoR results by the type of application, which would otherwise give interesting insight on the strengths and weaknesses of HLS. A qualitative analysis like that would also benefit from access to the implementations' source codes, which are seldom available.

Table I shows a breakdown of the number of qualifying papers published each year. Because the number of papers from each year is low, it is not feasible to use our data to check for a possible trend in the QoR of HLS during these years. A longer year range would also be preferable for that kind of study.

III. Meta-Analysis

Table II gathers a summary of the metrics of interest and their frequency of occurrence in the reviewed papers. In general, the reviewed works have much variance in the reported details about the experimental setup and results. The table counts only those papers that report the results in exact terms either in absolute values or in percentages. Inexact values, such as "the execution time was less than 100 ms," were excluded from our quantitative analysis.

Twenty-two articles report results for more than one application or experimental setup. In many works, multiple different applications were implemented, often related to each

TABLE II
METRICS AND THE FREQUENCY OF THEIR OCCURRENCE IN THE REVIEWED PAPERS

Metric	Number of papers reporting (percentage of total)	Number of applications for which the metric was reported
HLS tool	42 (91%)	-
HLS input language	46 (100%)	-
Lines of input code	13 (28%)	36
Development time	15 (33%)	25
Maximum frequency	24 (52%)	74
Latency	10 (22%)	17
Execution time	8 (17%)	14
Performance	15 (33%)	46
FPGA LUTs/LCs/LEs/Slices	36 (78%)	92
FPGA Flip-flops	23 (50%)	63
FPGA DSP blocks	22 (48%)	50
FPGA BRAM	22 (48%)	55
ASIC area	4 (9%)	8
Power consumption	3 (7%)	7
Total papers	46	

TABLE III
HLS TOOL USAGE BY PAPERS

HLS Tool	N	Tool language
AccelDSP	1	MATLAB
Altera OpenCL	3	OpenCL
Bluespec	2	Bluespec language
C2RTL	1	C
Cadence Stratus	1	C/C++/SystemC
CAPH Toolset	2	CAPH language
Catapult-C	2	C/C++/SystemC
Chisel	2	Scala
Cadence C-to-Silicon	2	C/C++/SystemC
Convey Hybrid-Threading	1	HT language
HCE	2	C
HIPAcc	2	HIPAcc language
Impulse C	1	C
LegUp	3	C
MATLAB Simulink HDL Coder	1	MATLAB functions, Simulink models
Maxeler MaxCompiler	1	Java
ROCCC	1	C
Xilinx System Generator for DSP	2	MATLAB Simulink
Xilinx Vivado HLS/Autopilot	18	C/C++/SystemC
Undisclosed	4	N/A

other (for example, [9]–[11]). Some authors compared different HLS tools [12]–[14], whereas others compared various micro-architectural optimizations, such as loop unrolling and pipelining [15], [16] or different FPGA chips [17], [18]. The data set is thus larger than the mere number of qualifying papers would suggest. For brevity, we shall call each of these individual results *applications*, regardless of whether they are based on actual different applications, HLS tools, FPGA chips, or other variations. The third column of Table II shows the total number of applications for which the given metric is reported.

Development time is of great interest when comparing the HLS and RTL methodologies. However, only a third of the papers report the development time, which can be seen as a flaw in the articles omitting it. Of the various QoR metrics, FPGA resource usage is reported more often than the performance values. Only four papers target ASIC implementation (instead of FPGA), and thus there is not enough data to compare ASIC area results. The same is true for power consumption.

Almost all papers report the used HLS tool. The remaining works give no reason for not revealing the information, but license agreements may have been the cause. However, even those articles mention the HLS input language.

Table III shows a summary of the used HLS tools. The second column tells the number of occurrences of each tool and the third column their input languages. The table suggests that Vivado HLS (formerly known as Autopilot) is the most popular HLS tool, at least in academia. All the other tools gain only scattered usage. Vivado's popularity is probably due to Xilinx being the leading FPGA vendor, whose design suite for FPGAs includes Vivado HLS. The large number of used HLS tools also speaks of the relative immaturity of the field.

Of the 46 qualifying works, 39 used self-made RTL implementations for comparison with HLS and seven cited RTL results from other research groups. There are additional works that would have qualified for this paper, but they cite papers with incompatible RTL implementations, which resulted in their preclusion. For example, the FPGA chip used for RTL was from a different family, which prevented fair resource usage comparison.

IV. COMPARATIVE STUDY RESULTS AND ANALYSIS

A. On the QoR Metrics

The fundamental building block of FPGAs is a configurable logic block (CLB) or a logic array block (LAB), depending on the FPGA vendor and device. CLB/LAB consists of a few logical cells that may be called logic cells (LCs), logic elements (LEs), or adaptive logic modules. These logical cells are made of look-up tables and flip-flops. The reviewed papers usually report one of these figures when synthesizing an application for FPGA. For the purposes of this paper, it is irrelevant which figure was reported, since we are interested in the *ratio* of resource usage between HLS and RTL. Thus, we have grouped all of these resource metrics under the same term called *basic FPGA resources*.

FPGAs also contain other resources such as DSP blocks and on-chip block RAM (BRAM) memories, which cannot be converted to CLB equivalents without sufficient data from the FPGA vendors. This would require knowing the exact FPGA chip type, but only about 60% of the reviewed papers report it, and the others merely state the used FPGA family. Therefore, we had no universal way to combine all the resource metrics into a single resource usage value, which

TABLE IV
SUMMARY OF THE NUMERICAL DATA FROM THE PAPERS

Metric	N	HLS/RTL mean	Geometric std. dev.	HLS better or equal to RTL
Lines of code	36	**0.52**	2.26	75 %
Development time	25	**0.32**	2.59	88 %
Performance	46	0.47	5.50	39 %
Execution time	14	1.70	2.21	39 %
Latency	17	1.05	2.07	35 %
Maximum frequency	74	0.88	1.48	42 %
Basic FPGA resources	92	1.41	3.76	33 %
DSP blocks	50	1.11	-	68 %
BRAM blocks	29	**0.49**	-	45 %
BRAM (kB)	27	1.47	-	33 %

could be compared across applications. Thus, we discarded this approach and chose CLBs or its constituents as the basis for resource usage comparisons.

The reviewed papers also use various different performance metrics depending on the implemented application. These can be divided into four categories: 1) performance; 2) execution time; 3) latency; and 4) maximum frequency. In this context, performance can be interpreted in several ways depending on the application. For example, for a video encoder, it would mean frames per second, and for a cryptography module, it would mean encrypted bits per second. For applications with a clear start and finish, execution time is often reported, and some papers report latency, i.e., the number of clock cycles for processing a sample. The most often reported performance metric is the maximum frequency for which the application can be scheduled on the target FPGA.

We wanted to include as many performance metrics as possible so all of them are used in this paper. For papers that report more than one metric, we prioritize performance over execution time, execution time over latency, and latency over maximum frequency. Thus, we use only one of these values per application rather than try to create an arbitrary aggregate performance metric. In the figures of the following sections, we shall call the selected value just *performance*. We have also inverted execution time and latency values in calculations for the figures so that a larger value is always better. The way to examine the various data cloud figures in the following sections is not to compare individual data points to each other but to concentrate on the center of gravity and dispersion of the data.

B. Numerical Analysis

Table IV gathers the numerical aggregate data of our findings. *N* denotes the number of applications for which the corresponding data were reported. The third column reports the mean of the ratios between HLS and RTL results. For all the values except DSP blocks and BRAM, we used the geometric mean rather than the arithmetic one, since the values in each category can differ by orders of magnitude because of the wide variety of applications. For DSP blocks and BRAM, the geometric mean could not be calculated because of the zeros in the data set, so arithmetic mean was used instead. Bolded mean values favor HLS while unbolded values favor RTL. The fourth column shows the geometric standard deviation (GSD). Note that it is a multiplicative value: the lower bound is obtained by dividing by the GSD and the upper bound is obtained by multiplying by the GSD. The last column shows the percentage of results for which the HLS application performed as well or better than the corresponding RTL version.

TABLE V
COMPARISON OF QOR BY FRAMEWORK TYPE

	N	HLS/RTL performance ratio	N	HLS/RTL Basic resouce usage ratio
C based framework	100	0.64	71	1.26
Other frameworks	51	0.84	36	1.50

As expected, HLS outperforms RTL in both development time and lines of source code. The average development time is only about a third of a corresponding RTL application. We also examined the HLS to RTL *development time ratio* as a function of the *absolute development time* to see if the scale of the project had an effect on the ratio, but found no correlation. Thus, it seems that for both large-scale and small-scale applications the reduction in development time is the same. On the other hand, the respective comparison with code size shows that for larger applications (1000 LoC or more), HLS code seems to be more compact compared with RTL code. In fact, in all the cases where there was more HLS LoC than RTL LoC, the code size was less than 250 LoC. With smaller code size, nonbehavioral code takes a relatively larger part of the total, which seems to favor RTL.

In performance and execution time, the HLS design is on average clearly inferior, but in latency and maximum frequency the difference is less prominent. The HLS approach also loses in basic resource usage: On average, HLS uses 41% more basic FPGA resources than RTL. With BRAM and DSP blocks, the results are ambivalent. Based on papers, which report the number of used BRAM blocks, HLS seems to use them more efficiently, but with papers, which report BRAM usage in kilobits, RTL wins. In DSP block usage, HLS and RTL seem similar.

We also examined how the HLS input language affects the QoR. In [19], the HLS tools are divided into five categories based on their style of describing the input: hardware description language (HDL) like frameworks, C-based frameworks, high-level language (HLL)-based frameworks (these are highly abstract, usually object-oriented languages), model-based frameworks (using executable specification, e.g., NI LabView and MATLAB HDL Coder), and CUDA/OpenCL-based frameworks. In this paper, we found five applications implemented with HDL like, 77 with C-based, 10 with HLL-based, six with model-based, and 11 with CUDA/OpenCL-based frameworks. Since other than C-based frameworks receive only scattered usage, it is not prudent to compare all the categories with each other. Instead, we compare the QoR of C-based frameworks and all the others. The results are shown in Table V, where *N* denotes the number of

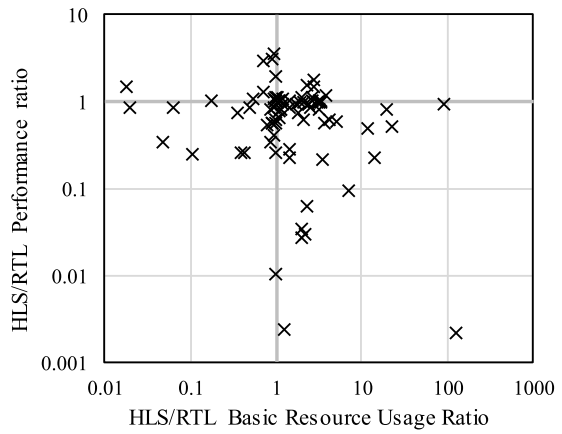


Fig. 1. Scatter graph of the HLS to RTL ratio between performance and basic resource usage for different applications.

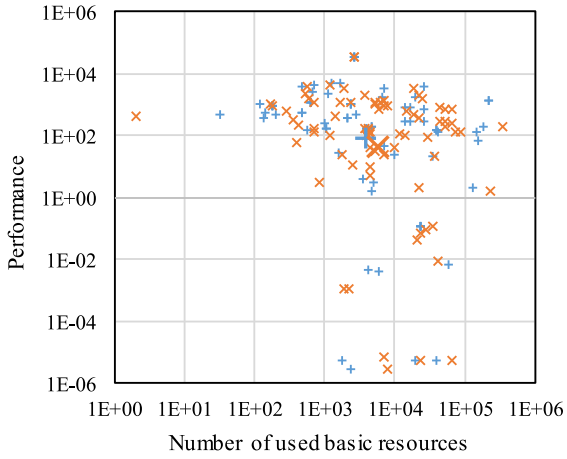


Fig. 2. HLS (orange x) and RTL (blue $+$) performance and basic resource usage for each application. Note that the performance does not have a listed unit as it varies from application to application.

comparable results. It seems that C-based frameworks produce designs with worse performance than the other frameworks but save in basic resource usage. Looking further into the data, we noticed that the CUDA/OpenCL-based frameworks were especially resource consuming (3.56×) and produced the worst performance (0.56×).

C. Comparisons Between Resource Usage and Performance

To better illustrate the QoR differences, Fig. 1 shows the relative HLS/RTL performance against the relative HLS/RTL basic resource usage for each application. Each "X" in the figure represents a single application. The wider horizontal and vertical lines denote break-even lines where the performance and basic resource usage are the same for both HLS and RTL, respectively. Most of the marks are clustered around the intersection of the break-even lines, indicating that in the great majority of cases the performance and basic resource usage difference between HLS and RTL is relatively small. Nevertheless, there are more marks toward the right and bottom of the figure than in the opposite directions, showing that RTL tends to outmatch HLS in both regards.

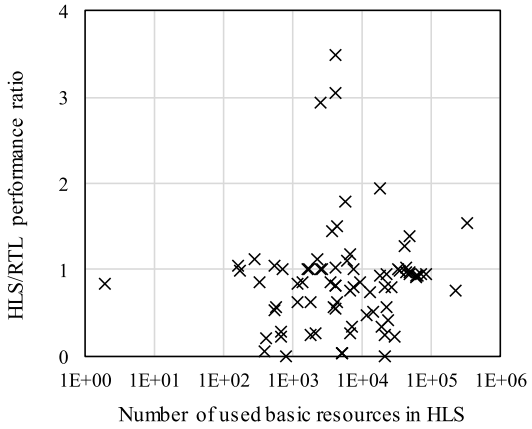


Fig. 3. HLS/RTL performance versus HLS basic resource usage.

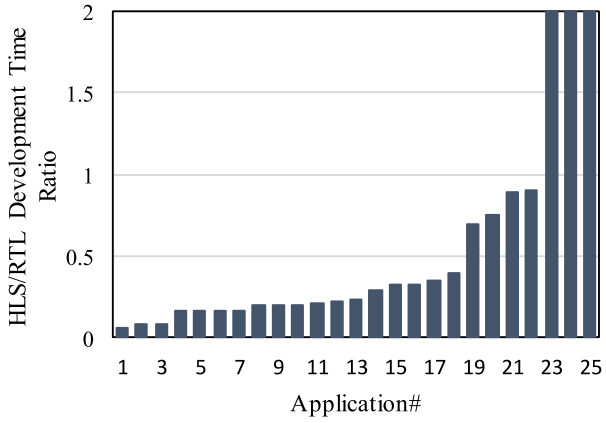


Fig. 4. HLS/RTL development time ratio for different applications.

Another way to look at the same data is depicted in Fig. 2, which shows the absolute performance and basic area usage values for both HLS applications ("+") and RTL applications ("x"). The large, partially overlapping symbols show the centers of gravity based on geometric means for both metrics correspondingly. The data point clouds are largely overlapping, and the centers of gravity lie close to each other. Thus, on average there is no radical difference between the HLS and RTL QoRs, but RTL fares somewhat better.

We also wanted to see, whether there exists any correlation between the relative HLS/RTL performance and the absolute numbers of basic resource usage. That is, does the relative performance between HLS and RTL designs change as a function of consumed FPGA resources. Our hypothesis was that the HLS tools' ability to optimize data path and control logic might be more limited with larger applications. The results are plotted in Fig. 3, which shows that there is no clear correlation, and indeed, the Pearson correlation coefficient is only 0.10 for this data set. Thus, the size of the design does not seem to affect the HLS tools' ability to optimize performance.

D. Comparisons Based on Design Effort

Fig. 4 shows the HLS/RTL development time ratio for applications for which the development time was reported. In all but three cases, the ratio is less than one, and in 72% of cases, it

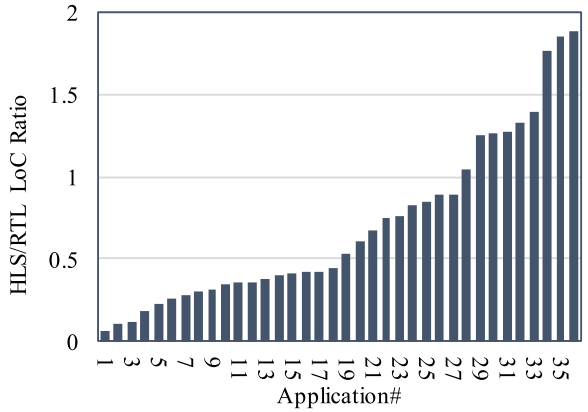
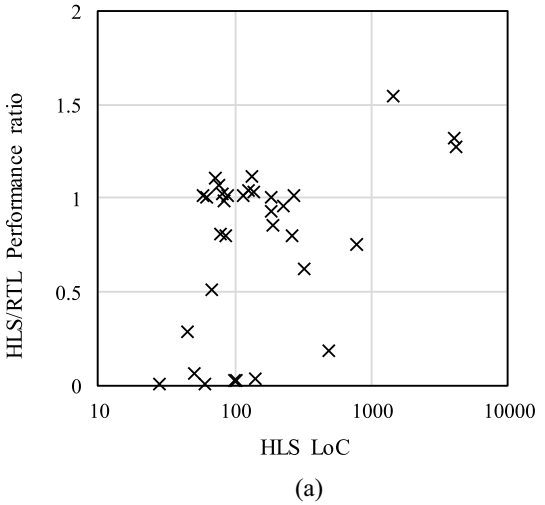


Fig. 5. HLS/RTL LoC ratio for different applications.



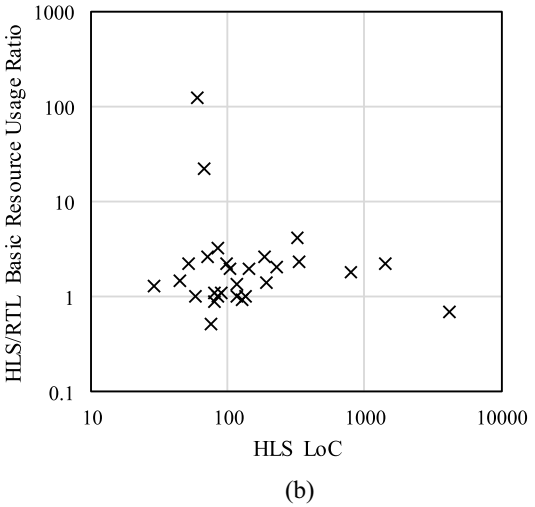


Fig. 6. HLS/RTL (a) performance ratio (b) basic resource usage ratio versus HLS LoC.

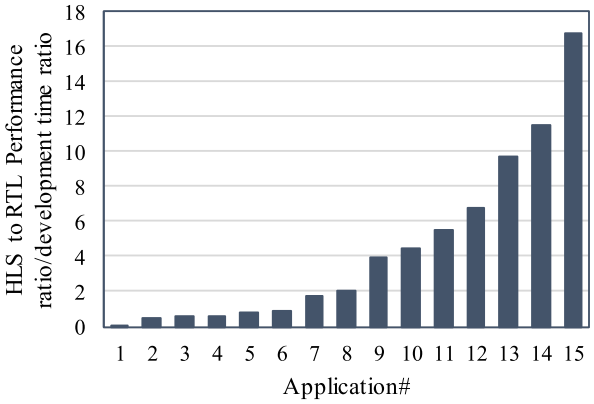


Fig. 7. Relative HLS to RTL productivity for different applications.

is less than 0.5. The three applications, where the HLS development time is larger than that of RTL, are from the same work [13]. The authors stated that the difference in development time was due to the time spent to learn to use the HLS tool and the need to modify the reference C++ source code to reach the required throughput.

Similarly, Fig. 5 depicts the LoC ratio between HLS and RTL designs. Here, the HLS dominance is less prominent but still significant. In 75% of cases, the HLS LoC is smaller than RTL LoC.

We also investigated a possible correlation between the size of the application in LoC and HLS/RTL performance. Fig. 6(a) shows the data. When the three outliers in the top right corner are eliminated from calculations, the Pearson correlation coefficient is only 0.04. Thus, it seems that the size of the code is no indication for the relative HLS/RTL performance. Fig. 6(b) shows the same data for the relative HLS/RTL basic resource usage. The correlation is −0.08, so the code size does not correlate with the basic resource usage ratio either. Taken together, Figs. 3 and 6 indicate that the complexity of the application has no effect on the relative HLS to RTL performance or basic resource usage. However, as Fig. 6 shows, the majority of the applications presented in the papers are rather small in terms of LoC. Studying the respective behavior with larger applications is omitted due to the absence of data.

One way to look at the usefulness of HLS relative to RTL is to examine the performance obtained per design hour as discussed in [11]. Fig. 7 shows the relative productivity for all applications for which both performance and development time is reported, by dividing the HLS/RTL performance by the development time ratio. A value larger than one indicates that the HLS approach gives more performance per design hour than RTL. The average value is 4.4. RTL approach clearly wins in cases 1 to 4. The methodologies are about equal in cases 5 and 6, and HLS is the better approach in the remaining cases. For application 1, the bar is almost invisible, as the ratio is 0.05. This application is a sparse algorithm matrix multiplication [11] with dynamic loop bounds, which are unsuitable for the automatic optimizations that HLS tools perform to speed-up computation. Despite that, the figure indicates that on average, a designer gets more performance per design hour with HLS tools.

V. Test Group Study

This survey demonstrates that many prior works report the results of HLS to RTL comparisons rather inadequately, which

TABLE VI
BACKGROUND EXPERIENCE OF THE TEST GROUP

Person #	LoC (SW)	SW experience (months)	LoC (HW)	HW experience (months)
1	1k	18	1k	10
2	10k	3	1k	3
3	10k	18	1k	3
4	100k	50	1k	0
5	10k	3	1k	3
6	1k	1	1k	1

also complicated our data collection. Therefore, we organized a case study to demonstrate the best practices in setting up appropriate tests for such comparisons and reporting the results. A secondary purpose of the case study was to examine how HLS and RTL flows differ from the user's perspective and what is the relative productivity of the flows. Most previous studies have focused only on the QoR differences instead.

The case study was to implement a 2-D discrete cosine transform (DCT) [20] algorithm for 8×8 residual blocks used in the high efficiency video coding (HEVC) [21] encoder. DCT was chosen because it is well known and of suitable complexity for this paper.

A. Test Group

The test group consisted of six participants having basic knowledge on digital design and programming. As shown in Table VI, they had written 1k to 100k lines of C or C++ previously. On average, they had about 15 months of programming experience in work or hobby projects. The participants were much less experienced in hardware design with an average of 1k lines of VHDL or Verilog code and three months of experience in such projects. Only one of the participants had done a small tutorial with HLS before this study, making this experiment the first introduction to HLS for the rest.

We selected participants with limited hardware experience but moderate software experience, as HLS promises to hide away the hardware-specific implementation details. Thus, programmers who are used to writing behavioral descriptions in software projects are an ideal audience for HLS. Indeed, the litmus test of HLS is that such users reasonably effortlessly can produce acceptable results when designing relatively simple hardware blocks.

To acquire sufficient background knowledge of HLS, the participants self-studied the HLS basics and carried out five small exercises implementing parts of an audio codec for FPGA. Previously, they had done the same exercises using VHDL RTL.

B. Test Case

In an HEVC encoder, DCT is used to convert 8×8 spatial-domain residual blocks into 8×8 transform-domain coefficient (tcoeff) matrices. A well-known row-column algorithm [20] executes these 2-D transforms with separable 1-D transforms in two consecutive stages. The transform is first applied to each row of the residual block to generate an intermediate matrix and then to each column of the intermediate matrix to generate the final transform coefficient matrix.

The participants were assigned to implement this 2-D DCT hardware unit for 8×8 residual blocks with RTL (VHDL or Verilog) and HLS (C/C++ with Mentor Graphics' Catapult-C version 8.2 m UV). Catapult-C supports the whole design flow from writing the original source code to generating and verifying the RTL code. In this paper, no physical FPGA implementation was made, but only the synthesis results were used to obtain the QoR data. Performing place & route (P&R) was omitted as we were interested in the relative HLS to RTL results, and P&R should not affect the ratio significantly.

The provided DCT references included the HEVC specification and its implementation in the HEVC reference encoder [22]. The participants were also given a ready-made SystemC test bench and requirements for the interfaces to make the test bench work without modifications. Interface requirements included the widths of the input and output data buses and related control signals. The same test bench was used for the RTL and HLS versions. It generated random residual values for the first pass and performed the necessary transpose for the second pass. The condition for successful implementation was to pass the test bench validation.

The participants were also instructed to allocate their working hours to five categories: 1) designing; 2) implementing; 3) searching information; 4) simulating; and 5) debugging. They were allowed to choose whether to implement the HLS or RTL version first or both simultaneously.

C. Results

Table VII shows the area and speed figures of the RTL and HLS implementations for the individual test persons. The HLS/RTL ratio shows the ratio between the results for HLS and RTL. The bolding indicates when the HLS flow achieved better results. The speed was calculated as million transform coefficients per second using the output coefficients, latency, throughput, and frequency reported by the participants.

Four test persons started the work with the RTL implementation. All participants wrote the RTL code with VHDL rather than Verilog. Even though the smallest area and the highest frequency were achieved with RTL, the overall trend was that the participants were able to get slightly smaller area and slightly higher clock frequencies with the HLS tool. Furthermore, the HLS designs are over $2.5\times$ as fast as the RTL designs, which also affected the speed to area ratios. For example, person #4 achieved the best speed to area ratio of all designs with HLS. On the other hand, person #3 was the only one who got better speed to area ratio with RTL. All test persons used a multistage structure to calculate the DCT in RTL code, but none of them implemented a more complex state machine to use stage pipelining for consecutive inputs. The lack of pipelining lowered throughputs in the RTL case. In comparison, all persons were able to use loop unrolling and pipelining in HLS to achieve much better throughput values than with RTL.

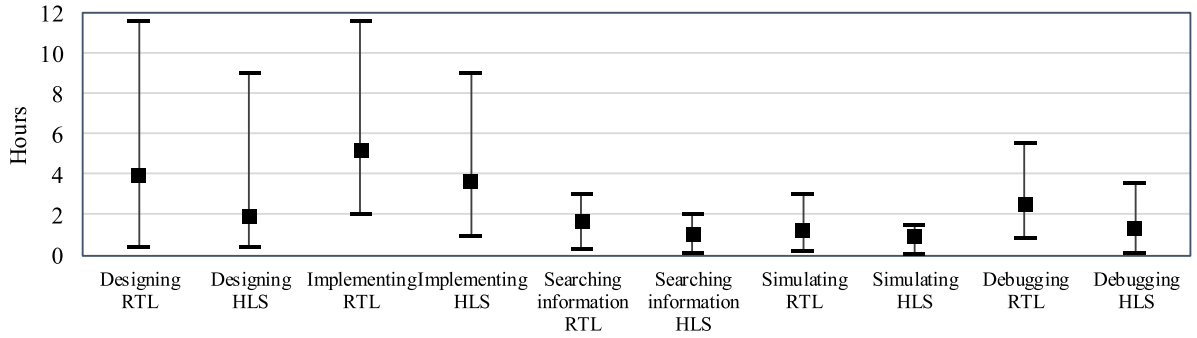


Fig. 8. Maximum, minimum, and average time usage for different categories with RTL and HLS.

TABLE VII
AREA AND PERFORMANCE FIGURES OF RTL AND HLS DESIGNS

HLS Person #	Area (LUTs)	Freq. (MHz)	Speed*	Speed/Area**	Started
1	1 860	214	258	139	
2	3 161	101	588	186	x
3	2 814	211	675	240	
4	2 273	167	972	427	x
5	2 768	137	797	288	
6	2 463	211	750	305	
Avg. RTL	2 557	174	673	264	
1	4 000	145	197	49	x
2	2 068	108	192	93	
3	1 292	308	458	355	x
4	4 431	148	499	113	
5	2 066	137	122	59	x
6	2 722	149	121	44	x
Avg. HLS/RTL ratio	2 763	166	265	119	
1	**0.47×**	**1.48×**	**1.31×**	**2.81×**	
2	1.53×	0.94×	**3.06×**	**2.00×**	
3	2.18×	0.69×	**1.47×**	0.68×	
4	**0.51×**	**1.13×**	**1.95×**	**3.80×**	
5	1.34×	1.00×	**6.55×**	**4.89×**	
6	**0.90×**	**1.42×**	**6.22×**	**6.87×**	
Avg.	**0.93×**	**1.05×**	**2.54×**	**2.22×**	

*Mtcoeffs/s **Mtcoeffs/kLUTs

TABLE VIII
HLS AND RTL PRODUCTIVITY

	HLS		RTL		HLS/RTL
Person #	Hours	Quality*/Hours	Hours	Quality*/Hours	Quality Ratio
1	2	80	4	14	5.9×
2	4	44	9	11	4.1×
3	12	19	21	17	1.1×
4	9	47	18	6	7.6×
5	5	60	9	6	9.4×
6	20	15	26	2	9.0×
Avg.	9	44	14	9	6.2×

*Mtcoeffs/kLUTs

Table VIII tabulates productivity values for HLS and RTL approaches. The productivity of all participants was clearly better with the HLS tool, and the average productivity of HLS was up to 6.0 times that of RTL. Hence, it is even higher than that found in the survey results. We can speculate how the productivity would have changed if the persons had implemented stage pipelining in their RTL implementations. It is still unlikely that the productivity levels had shifted to support RTL over HLS, as the time usage would have increased along with the throughput.

Fig. 8 shows the time usage of the participants in five categories. On average, the persons used less time within all categories when working with HLS. The grand total for maximum, average, and minimum time usages with the RTL flow was 37.7, 15.1 and 3.7 h, respectively, whereas the same values for the HLS flow were 25.0, 10.1, and 1.6 h.

As a conclusion, all participants had better productivity with HLS than with RTL. Although the group size was small, and the hardware background of the persons was very similar, this study shows that it is easier to adopt HLS than RTL and receive better results faster for people who have most of their experience in software design. This result underlines the fact that HLS is a useful tool for software engineers who want to implement, for example, hardware accelerators.

It should be noted that our result differs from the typical surveyed study, where the QoR of RTL was better than that of HLS. The likely explanation for this is that in the surveyed works, the designers had significantly more previous hardware expertise than our test persons. On the other hand, our case study is in line with the surveyed literature concerning productivity, which favors HLS.

D. Feedback From the Test Persons

After completing the test assignments, the participants were asked about the pros and cons of HLS and RTL design flows, out of which they finally had to select their favorite. The answers were split evenly (3-3) between HLS and RTL flows.

The persons favoring RTL over HLS hoped for more open source support for HLS tools, as the flow is highly tool dependent. This would allow more hobbyists to use HLS tools. Some test persons also wished for more control in the HLS tool over the resulting RTL in terms of cycle accuracy. For them, RTL was easier to fine tune and it gave them a better understanding of the problem at hand.

The persons favoring HLS over RTL liked the ease of HLS, where unnecessary details such as automatic I/O handshaking and pipelining support can be left as the responsibility of the HLS tool. This let the participants to focus on defining the behavioral description. They also felt that RTL was more

time consuming, required more planning, and would have been harder to redesign.

The overall conclusion from the test persons was that HLS versus RTL compares to C versus assembly languages in embedded programming. They expressed the view that a designer would rather use HLS, the highest level of abstraction available, and the lower level RTL should only be used in cycle critical applications or if it is able to provide a noticeable increase in performance.

E. Best Practices

We use our literature survey and this case study to sum up the following best practices for conducting comparative studies with RTL and HLS work flows.

1) A group of individuals should be used to implement the same design to lessen the impact of designer experience with the two flows.
2) The used HLS tools and languages should be reported unless license agreements prevent that. These choices have been shown to affect the QoR [12]–[14].
3) The same microarchitectures should be used in both RTL and HLS designs when conducting a study that concentrates on the QoR differences. If the emphasis is on productivity or the usability of HLS, however, then this restriction can be lifted.
4) For FPGA implementation, the exact FPGA chip model and version should be reported to allow replication of the results.
5) The time usage by each designer should be reported. Additionally, the time spent in each work phase should be reported to allow more insight into what parts are the most time consuming with HLS and RTL versions.
6) Lines of input code should be reported to show the size and complexity of the applications.
7) In addition to the basic QoR results, performance per design time should be reported to show the difference in productivity between the HLS and RTL flows.

VI. Closing the Quality Gap

Our survey shows that more often than not, there still is a QoR gap between the HLS and RTL methods for any given application, usually favoring RTL. A large amount of literature exists that has recognized the gap and proposes ways to close it. In this section, we present a survey of that literature to highlight it for the HLS researchers and developers. In addition, we review papers that introduce novel improvements to the existing HLS flows.

A. Research Directions for Tool Developers

Sun *et al.* [8] have several suggestions for the HLS tool developers for focusing their efforts. They note that resource sharing and scheduling are two major features in HLS techniques that the current HLS tools still struggle with. For example, they demonstrate that an HLS tool instantiates 31 hardware operators of a certain type when only 13 would be needed with optimal sharing. They also note that the HLS tools obfuscate the relationship between the source code and the generated hardware, which in turn makes it hard to identify the suboptimal parts of the code. Furthermore, the authors call for the industry to agree on a standard C-based input language for HLS. This would allow an unambiguous way for the tool users and the tools themselves to interpret the source code.

Rupnow *et al.* [57] recognized room for development in both the usability and the QoR of HLS. Their study uses AutoPilot (now Vivado HLS), but the advice is generalizable. The authors propose automatic tradeoff analysis of loop pipelining and unrolling to make the DSE faster. With complicated loop structures, the number of possible optimization combinations can be very high. In addition, the authors call for support for BRAM port duplication directives, more robustness for dataflow transformations, and support for streaming computation for 2-D access patterns. To improve the QoR, they suggest that the tools should detect memory level dependence between separate loops and functions, and automatically reorder memory access to allow partitioning, streaming, and better pipelining. The tools should also automatically create buffers to improve memory access reuse.

The importance of optimized memory accesses in high-quality designs is also recognized in [5]. The authors point out that the HLS tools usually do not have support for memory hierarchy nor do they abstract external memory accesses. Therefore, the designer is required to pay attention to the details of bus interfaces and memory controllers, which does not sit well with the idea of behavioral design paradigm. The HLS tools should hide external memory transfers from the designers to fix this problem. This paper also notices the difficulty of obtaining task-level parallelism from sequential C/C++ specifications, for which the authors suggest developing an appropriate device-neutral programming model.

In [32], the lack of support for dynamic data structures in HLS tools is brought forward. The authors implement the same algorithm with a data-flow centric way and by using recursive tree traversal, which uses dynamic memory allocation, and observe a significant performance reduction using the latter method. By applying several manual code transformations, the authors can increase the performance, and conclude that the HLS tools should automatically perform similar optimizations with dynamic data structures.

B. Improvements to the HLS Flow

Since the writers of research articles typically have no access to the source code of commercial HLS tools, most papers that have improved upon the HLS results do so by introducing new optimization steps to the design flow. Some promising results falling in this category are reviewed in this section.

Josipovic *et al.* [58] proposed using parallel pattern templates to scale module implementation according to the properties of the target device, exceeding the capability of the HLS tool to do so. The authors show up to 2.8× speed-up over a standard HLS tool flow. A template-based approach is also used in [59], where composable and parameterizable templates of common computation patterns optimized for hardware are

used to improve performance. These kind of templates could be included in HLS tools for the users' convenience.

In [60], the problem of memory access bottleneck in massively parallel algorithms is discussed. The authors propose an algorithm that schedules the memory accessed during different pipeline stages reducing the simultaneous access pressure. Their approach improves the pipelining performance by 43% on average and reduces memory bank usage by 55%. Another way to reduce memory access overhead is discussed in [61]. This paper presents a novel algorithm to scalarize arrays selectively to on-chip registers within certain area constraints. The results indicate significant performance improvements.

One method to enable more efficient HLS is by identifying custom operations that are merged from sequential basic operations. This reduces the complexity of the data flow graph of the synthesized algorithm, which in turn reduces synthesis runtime and improves the QoR. This method has been studied in [62], achieving significant improvements in area consumption, performance, and code size. Therefore, HLS tools should include custom operation identification as a preprocessing step.

Resource allocation and operation binding are two of the basic steps in HLS. Thus, their efficient implementation is of critical importance in achieving good QoR. In [63], the effect of register allocation has been investigated. This paper shows that in most cases a naïve resource allocation strategy, i.e., one register per variable without register sharing leads to the best QoR results.

HLS tools use a software compiler to create an intermediary representation (IR) of the input program. The IR is then used in the HLS optimization steps. It is not surprising that the IR and thus the compiler options affect the QoR. Huang *et al.* [64] have studied the effect of different compiler options on the QoR and developed a method to automatically select only those options that improve the QoR, achieving on average a 16% better performance compared to the usual –O3 optimization level.

In [65], it is observed that significant area savings can be achieved by merging different behavioral descriptions instead of performing HLS for each of them separately. This is due to allowing better resource sharing of functional units on FPGA when the HLS tool can share them between descriptions. This paper presents an algorithm for searching for optimal mergings within given latency constraints.

C. Design Space Exploration

The HLS tools contain various directives that can guide the hardware synthesis to generate more efficient designs. These directives include pipelining and unrolling of loops and array partitioning among others. Since most algorithms contain numerous loops and data arrays, finding the group of Pareto optimal directive settings can be a daunting task, yet it is essential for good QoR. Exploring the design space for optimal settings should therefore be automated, but currently the leading HLS tools do not help the user in DSE. On the other hand, there are a few academic papers that have studied the DSE automation in HLS.

A straightforward automated iterative DSE methodology is presented in [66]. The method, which focuses on area reduction, achieves up to 50% improvement in the QoR when compared to nonguided HLS flow. A more complicated DSE algorithm, based on an adaptive windowing method, is shown in [67]. This algorithm is shown to offer a good tradeoff between running time and finding the best QoR. A similar approach specializing on applications with nested loops has demonstrated up to 235× speed-up compared to exhaustive DSE, while achieving similar results [68].

A sequential model-based optimization has been applied to the DSE problem in [69]. This paper shows that the method can find globally optimal points from a space of tens of thousands of possible designs in reasonable time. In [70], a lightweight preprocessing step has been added before HLS to perform dynamic dependence analysis of the target algorithm. The method can expose resource sharing opportunities that result in better QoR when they are given as constraints to the HLS tool.

The specific but important question of finding the optimal loop unrolling factor has been discussed in [71]. The authors have developed an algorithm for finding the optimal unrolling factor within given area constraint and show that it can provide the best performance compared to other possible solutions.

D. Verification

Verification remains a time-consuming part of any design project. Therefore, it is crucial that the HLS tools support the verification flow on all stages. While the HLS flow allows for efficient behavioral verification of single modules, the generated RTL must still be verified for nonbehavioral aspects such as interface synthesis results and successful component integration. Traditional RTL verification after HLS is difficult since there is no direct relationship to the input source code [4], [5], [8]. Nevertheless, verification time has been halved in many cases using HLS [72].

The verification aspects of HLS have been extensively discussed in a recent paper [72]. The author points out that logic redundancy, which lowers test coverage, is a major problem with HLS. Logic redundancies may be present in the source specification but also introduced by the HLS tool in the RTL generation. Thus, the developers of the HLS tools should strive to eliminate the tendency to generate logic redundancy. Besides that, formal tools can be used to identify the redundancies during verification. This paper also promotes source linting as a way to improve HLS. Not only can it be used to check for error sources but also to help with the design optimization by proving properties such as FIFO sizes.

Cong *et al.* [5] presented three noteworthy items to enable most of the debugging to occur on the behavioral input language level for on-chip validation: 1) the ability to add debugging logic with small overhead; 2) the ability to observe critical buffers such as FIFOs; and 3) the ability to observe the internal states of hardware blocks using breakpoints in the source code. These important debugging features cannot be implemented on the RTL level after performing HLS because of the machine-generated RTL code.

TABLE IX
SUMMARY OF THE REVIEWED PAPERS

[#]	Year	HLS tools	Modules or algorithms	Number of applications	LoC	Dev. time	Performance	Basic FPGA Resources
[5]	2011	AutoPilot	Multi-I/O sphere decoder	1		x		x
[8]	2016	Undisclosed	AES encryption	1	x		x	
[9]	2014	Catapult-C	K-means accelerator, histogram map/reduce, matrix mult., word count	5			x	x
[10]	2016	Vivado	Data pinning, step row filter, Sobel filter	3		x	x	x
[11]	2013	Vivado	Matrix multiplication	3		x	x	x
[12]	2015	Vivado, LegUp, Simulink HDL	HEVC 2D IDCT	3			x	x
[13]	2014	Vivado, Catapult-C	Predictive block-based motion estimation	8	x	x	x	x
[14]	2014	Altera OpenCL, BlueSpec, Chisel, LegUp	Bitonic sorter, spatial sorter, median operator, hash join	16	x		x	x
[15]	2016	Cadence C-to-Silicon	Semi-global matching algorithm for stereo vision	4			x	x
[16]	2011	Xilinx System Generator for DSP, Simulink HDL	Hybrid RAKE receiver for DS-UWB communication	2				x
[17]	2011	Xilinx System Generator for DSP, Simulink HDL	Adaptive impulsive noise filtering	3			x	x
[18]	2014	Vivado	128-bit key AES-CTR cryptography	6			x	x
[23]	2010	BlueSpec, unspecified	Reed-Solomon decoder for 802.16 protocol receiver	2			x	x
[24]	2010	ROCCC 2.0	Ten small separate applications	10	x	x	x	x
[25]	2010	Impulse-C	Computed tomography filtered backprojection	2		x	x	
[26]	2010	AccelDSP	Image preprocessing coprocessor	1		x	x	x
[27]	2011	HCE	Reverse time migration for solving a wave equation	1	x	x	x	
[28]	2011	AutoESL AutoPilot	Sphere detector for spatial multiplexing MIMO	1		x		x
[29]	2012	Cadence C-to-Silicon	Hardware-based run-time monitors	4	x		x	
[30]	2012	HCE	GRN generator for Brownian motion simulation	1		x	x	x
[31]	2012	Chisel	3-stage 32-bit RISC processor	2	x			
[32]	2013	Vivado	K means clustering	2			x	x
[33]	2013	Vivado	Operating system scheduler	1			x	x
[34]	2013	Vivado	Buck converter closed-loop control	1			x	x
[35]	2013	Catapult-C	IEEE 802.15.4 physical layer for SW defined radio	1				x
[36]	2013	MaxCompiler	LDPC decoder	1			x	
[37]	2013	Vivado	Skin detection image processing system	1				x
[38]	2013	Vivado	Convolution, background subtraction	2		x	x	x
[39]	2014	Altera OpenCL, Vivado	Tri-diagonal linear system solver	2			x	x
[40]	2014	Vivado	10 Gb/s network flow monitor	1	x	x		x
[41]	2014	Undisclosed	Floating-point unit co-processor	3			x	
[42]	2014	CAPH Toolset	MPEG encoder core	1	x		x	x
[43]	2014	Vivado	Memcached key-value store	1	x		x	x
[44]	2014	CAPH Toolset	Histograms of oriented gradients	1	x			x
[45]	2015	Undisclosed	Microwave imaging via space-time beamforming	1			x	
[46]	2015	LegUp	MIPS ISA processor	2			x	x
[47]	2015	Convey Hybrid-Threading	Sobel edge detector, breadth-first search, Smith-Waterman sequence alignment	3	x	x	x	x
[48]	2015	Altera OpenCL	Canny edge detector, Sobel edge detector, SURF feature extraction	6		x	x	x
[49]	2015	HIPAcc & Vivado	Stereo vision block matching	1			x	x
[50]	2015	HIPAcc & Vivado	Stereo block matching	2			x	x
[51]	2015	Vivado	Non-binary LDPC decoder	1			x	
[52]	2016	Cadence Stratus HLS	Direct memory access controller	1	x	x	x	
[53]	2016	Vivado	HEVC sub-pixel interpolation	1			x	x
[54]	2016	Vivado	HEVC intra prediction	1			x	x
[55]	2016	C2RTL	Bilateral filter for image denoising	1			x	x
[56]	2016	Vivado	SURF algorithm	1			x	x

The "x" marks denote what information was given in the papers

Besides verification, engineering change orders (ECOs) present difficulties with HLS [4]. When an ECO is issued, only some small incremental changes are required, which are typically not captured by the high-level behavioral description. On the other hand, it has been noted that since the behavioral source code can be extensively verified and the HLS tools ensure that the generated RTL is correct, ECOs are uncommon in HLS flows [66].

VII. CONCLUSION

In this paper, we examined 46 recent articles about comparisons on the QoR and design effort between HLS and RTL design flows. As HLS promises great productivity gains over RTL, our aim was to see whether the contemporary HLS tools are also able to produce results that can compete with hand tuned RTL designs.

Our survey indicates that even the newest generation of HLS tools does not provide as good performance and resource usage as manual RTL does. However, there is a great variance in the results and HLS is shown to equal or outperform RTL approach in about 40% of the evaluated cases. Our own case study demonstrates that a designer with limited hardware experience can obtain better results with HLS, with 2.5 times more performance and slightly lower FPGA resource usage. We also examined whether the size of the design affected the relative QoR between HLS and RTL, but found no correlation. Thus, HLS seems as suited for small as large designs.

In design effort, the survey showed that HLS was clearly the frontrunner as expected. On average, the HLS design time was only a third of the corresponding RTL design time. In addition, the size of the HLS input code was almost halved, being 52% of the RTL code size on average. When taking into account both the QoR and the design effort, we found out that a designer gets on average 4.4 times as much performance per design hour using HLS than RTL. Our own case study supported this argument by reporting 6.0 times increase in productivity. Thus, HLS is a particularly good choice when time to market is a dominant issue and there is no compelling need to gain the ultimate performance or smallest resource usage for the product. HLS also offers tremendous time savings when architectural changes are made to an existing design.

In our reference literature, there was often lacking information, which made the HLS to RTL comparisons more challenging. Therefore, our case study also demonstrated the best practices in reporting HLS and RTL results for the same application. Preferably, the test group should be larger than we had at our disposal, but our test case still shows the essential details that we recommend reporting in this kind of research. In the future, a similar case study could be carried out with a test group with more hardware expertise. While this paper shows that people with limited hardware experience can easily adopt HLS and produce good results, it would also be interesting to see how the productivity and QoR differences behaved with hardware engineers as test persons.

Verification effort comparisons were also often missing from the surveyed papers. Most often, there was only a brief note on how HLS tools allow convenient use of behavioral test bench in RTL verification. As verification is a major part of any hardware project, this is a significant oversight in the state of HLS research. Therefore, in the future, more quantitative studies should be carried out on HLS versus RTL verification flows.

We also surveyed the literature for both suggestions and completed research for improving the QoR and verification flow of HLS. We found numerous papers that showed methods to improve the QoR significantly by adding new steps to the HLS design flow or by automating the DSE.

With the progress achieved in HLS tools during the last decade, we can conclude that the methodology is ready for adoption by the industry in prototyping and fast product development. If the next generation of HLS tools can close the QoR gap entirely, then HLS will become the new standard design method, and RTL can be targeted at similar limited fine-tuning as assembly languages in software development today.

APPENDIX

See Table IX.

REFERENCES

[1] *International Technology Roadmap for Semiconductors, 2011 Edition, Design*, ITRS, 2011, [Online]. Available: https://www.dropbox.com/sh/r51qrus06k6ehrc/AACQYSRnTdLGUCDZFhB6_iXua/2011Chapters?dl=0&preview=2011Design.pdf

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

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

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

[5] J. Cong *et al.*, "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.

[6] S. Windh *et al.*, "High-level language tools for reconfigurable computing," *Proc. IEEE*, vol. 103, no. 3, pp. 390–408, Mar. 2015.

[7] R. Nane *et al.*, "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.

[8] Z. Sun *et al.*, "Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis," in *Proc. 21st Asia South Pac. Design Autom. Conf.*, Macau, China, 2016, pp. 218–225.

[9] M. Sharafeddin *et al.*, "On the efficiency of automatically generated accelerators for reconfigurable active SSDs," in *Proc. 26th Int. Conf. Microelectron.*, Doha, Qatar, 2014, pp. 124–127.

[10] A. Cortes, I. Velez, and A. Irizar, "High level synthesis using Vivado HLS for Zynq SoC: Image processing case studies," in *Proc. Conf. Design Circuits Integr. Syst.*, Granada, Spain, 2016, pp. 1–6.

[11] S. Skalicky, C. Wood, M. Lukowiak, and M. Ryan, "High level synthesis: Where are we? A case study on matrix multiplication," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Cancun, pp. 1–7.

[12] E. Kalali and I. Hamzaoglu, "FPGA implementations of HEVC inverse DCT using high-level synthesis," in *Proc. Conf. Design Archit. Signal Image Process.*, Kraków, Poland, 2015, pp. 1–6.

[13] G. Schewior, C. Zahl, H. Blume, S. Wonneberger, and J. Effertz, "HLS-based FPGA implementation of a predictive block-based motion estimation algorithm—A field report," in *Proc. Conf. Design Archit. Signal Image Process.*, Madrid, Spain, 2014, pp. 1–8.

[14] O. Arcas-Abella *et al.*, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *Proc. 24th Int. Conf. Field Programmable Logic Appl.*, Munich, Germany, 2014, pp. 1–8.

[15] A. Qamar, F. B. Muslim, F. Gregoretti, L. Lavagno, and M. T. Lazarescu, "High-level synthesis for semi-global matching: Is the juice worth the squeeze?" *IEEE Access*, vol. 4, pp. 8419–8432, 2016.

[16] C. Thomos and G. Kalivas, "FPGA-based architecture and implementation techniques of a low-complexity hybrid RAKE receiver for a DS-UWB communication system," *Telecommun. Syst.*, vol. 52, no. 4, pp. 2115–2132, Apr. 2013.

[17] A. Rosado-Munoz, M. Bataller-Mompean, E. Soria-Olivas, C. Scarante, and J. F. Guerrero-Martinez, "FPGA implementation of an adaptive filter robust to impulsive noise: Two approaches," *IEEE Trans. Ind. Electron.*, vol. 58, no. 3, pp. 860–870, Mar. 2011.

[18] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Cancún, Mexico, 2014, pp. 1–8.

[19] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *ACM Queue*, vol. 11, no. 2, p. 40, Feb. 2013.

[20] M. Budagavi, A. Fuldseth, G. Bjøntegaard, V. Sze, and M. Sadafale, "Core transform design in the high efficiency video coding (HEVC) standard," *IEEE J. Sel. Topics Signal Process.*, vol. 7, no. 6, pp. 1029–1041, Dec. 2013.

[21] *High Efficiency Video Coding, Document ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC)*, document ITU-T Rec. H.265, ITU-T, Geneva, Switzerland, Apr. 2013.

[22] *Joint Collaborative Team on Video Coding Reference Software, Ver. HM 16.3*. Accessed: May 16, 2018. [Online]. Available: http://hevc.hhi.fraunhofer.de/

[23] A. Agarwal, M. C. Ng, and A. Kumar, "A comparative evaluation of high-level hardware synthesis using Reed–Solomon decoder," *IEEE Embedded Syst. Lett.*, vol. 2, no. 3, pp. 72–76, Sep. 2010.

[24] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proc. 18th IEEE Annu. Int. Symp. Field Program. Custom Comput. Mach.*, Charlotte, NC, USA, 2010, pp. 127–134.

[25] J. Xu, N. Subramanian, A. Alessio, and S. Hauck, "Impulse C vs. VHDL for accelerating tomographic reconstruction," in *Proc. 18th IEEE Annu. Int. Symp. Field Program. Custom Comput. Mach.*, Charlotte, NC, USA, 2010, pp. 171–174.

[26] M. Samarawickrama, R. Rodrigo, and A. Pasqual, "HLS approach in designing FPGA-based custom coprocessor for image preprocessing," in *Proc. 5th Int. Conf. Inf. Autom. Sustainability*, Colombo, Sri Lanka, 2010, pp. 167–171.

[27] T. Hussain, M. Pericás, N. Navarro, and E. Ayguadé, "Implementation of a reverse time migration kernel using the HCE high level synthesis tool," in *Proc. Int. Conf. Field Program. Technol.*, New Delhi, India, 2011, pp. 1–8.

[28] J. Noguera *et al.*, "Implementation of sphere decoder for MIMO-OFDM on FPGAs using high-level synthesis tools," *Analog Integr. Circuits Signal Process.*, vol. 69, nos. 2–3, pp. 119–129, Sep. 2011.

[29] M. Ismail and G. E. Suh, "Fast development of hardware-based run-time monitors through architecture framework and high-level synthesis," in *Proc. IEEE 30th Int. Conf. Comput. Design*, Montreal, QC, Canada, 2012, pp. 393–400.

[30] J. S. Malik, P. Palazzari, and A. Hemani, "Effort, resources, and abstraction vs performance in high-level synthesis: Finding new answers to an old question," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 5, pp. 64–69, Dec. 2012.

[31] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proc. 49th ACM/EDA/IEEE Design Autom. Conf.*, San Francisco, CA, USA, 2012, pp. 1212–1221.

[32] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *Proc. Int. Conf. Field Program. Technol.*, Kyoto, Japan, 2013, pp. 362–365.

[33] J. Dahlstrom, and S. Taylor, "Migrating an OS scheduler into tightly coupled FPGA logic to increase attacker workload," in *Proc. IEEE Mil. Commun. Conf.*, San Diego, CA, USA, 2013, pp. 986–991.

[34] D. Navarro, O. Lucia, L. A. Barragan, I. Urriza, and J. I. Artigas, "Teaching digital electronics courses using high-level synthesis tools," in *Proc. 7th IEEE Int. Conf. e-Learn. Ind. Electron.*, Vienna, Austria, 2013, pp. 43–47.

[35] V. Bhatnagar, G. S. Ouedraogo, M. Gautier, A. Carer, and O. Sentieys, "An FPGA software defined radio platform with a high-level synthesis design flow," in *Proc. IEEE 77th Veh. Technol. Conf.*, Dresden, Germany, 2013, pp. 1–5.

[36] F. Pratas, J. Andrade, G. Falcao, V. Silva, and L. Sousa, "Open the gates: Using high-level synthesis towards programmable LDPC decoders on FPGAs," in *Proc. IEEE Glob. Conf. Signal Inf. Process.*, Austin, TX, USA, 2013, pp. 1274–1277.

[37] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx vivado high level synthesis: Case studies," in *Proc. 25th IET Irish Signals Syst. Conf. China–Ireland Int. Conf. Inf. Commun. Technol.*, Limerick, Ireland, 2014, pp. 352–356.

[38] J. Hiraiwa and H. Amano, "An FPGA implementation of reconfigurable real-time vision architecture," in *Proc. 27th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, Barcelona, Spain, 2013, pp. 150–155.

[39] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Comparison of high level FPGA hardware design for solving tri-diagonal linear systems," *Procedia Comput. Sci.*, vol. 29, pp. 95–101, Jun. 2014.

[40] M. Forconesi, G. Sutter, S. Lopez-Buedo, J. E. Lopez de Vergara, and J. Aracil, "Bridging the gap between hardware and software open source network developments," *IEEE Netw.*, vol. 28, no. 5, pp. 13–19, Sep./Oct. 2014.

[41] C.-I. Chen, C.-Y. Yu, Y.-J. Lu, and C.-F. Wu, "Apply high-level synthesis design and verification methodology on floating-point unit implementation," in *Proc. Int. Symp. VLSI Design Autom. Test*, Hsinchu, Taiwan, 2014, pp. 1–4.

[42] J. Sérot and F. Berry, "High-level dataflow programming for reconfigurable computing," in *Proc. Int. Symp. Comput. Archit. High Perform. Comput. Workshop*, Paris, France, 2014, pp. 72–77.

[43] K. Karras, M. Blott, and K. Vissers, "High-level synthesis case study: Implementation of a memcached server," in *Proc. 1st Int. Workshop FPGAs Softw. Program.*, Munich, Germany, 2014, pp. 77–82.

[44] J. Sérot, F. Berry, and C. Bourrasset, "High-level dataflow programming for real-time image processing on smart cameras," *J. Real Time Image Process.*, vol. 12, no. 4, pp. 635–647, Dec. 2016.

[45] D. J. Pagliari, M. R. Casu, and L. P. Carloni, "Acceleration of microwave imaging algorithms for breast cancer detection via high-level synthesis," in *Proc. 33rd IEEE Int. Conf. Comput. Design*, New York, NY, USA, 2015, pp. 475–478.

[46] T. Ahmed, N. Sakamoto, J. Anderson, and Y. Hara-Azumi, "Synthesizable-from-C embedded processor based on MIPS-ISA and OISC," in *Proc. IEEE 13th Int. Conf. Embedded Ubiquitous Comput.*, Porto, Portugal, 2015, pp. 114–123.

[47] G. Wang, H. Lam, A. George, and G. Edwards, "Performance and productivity evaluation of hybrid-threading HLS versus HDLs," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, Waltham, MA, USA, 2015, pp. 1–7.

[48] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA," in *Proc. IEEE 26th Int. Conf. Appl. Specific Syst. Archit. Processors*, Toronto, ON, Canada, 2015, pp. 189–193.

[49] M. Schmid, O. Reiche, F. Hannig, and J. Teich, "Loop coarsening in C-based high-level synthesis," in *Proc. IEEE 26th Int. Conf. Appl. Specific Syst. Archit. Processors*, Toronto, ON, Canada, 2015, pp. 166–173.

[50] O. Reiche *et al.*, "Automatic optimization of hardware accelerators for image processing," in *Proc. DATE Friday Workshop Heterogeneous Archit. Design Methods Embedded Image Syst.*, Grenoble, France, 2015, pp. 10–15.

[51] J. Andrade *et al.*, "From low-architectural expertise up to high-throughput non-binary LDPC decoders: Optimization guidelines using high-level synthesis," in *Proc. 25th Int. Conf. Field Program. Logic Appl.*, London, U.K., 2015, pp. 1–8.

[52] Q. Zhu and M. Tatsuoka, "High quality IP design using high-level synthesis design flow," in *Proc. 21st Asia South Pac. Design Autom. Conf.*, Macau, China, 2016, pp. 212–217.

[53] F. A. Ghani, E. Kalali, and I. Hamzaoglu, "FPGA implementations of HEVC sub-pixel interpolation using high-level synthesis," in *Proc. Int. Conf. Design Technol. Integr. Syst. Nanoscale Era*, Istanbul, Turkey, 2016, pp. 1–4.

[54] E. Kalali and I. Hamzaoglu, "FPGA implementation of HEVC intra prediction using high-level synthesis," in *Proc. IEEE 6th Int. Conf. Consum. Electron.*, Berlin, Germany, 2016, pp. 163–166.

[55] I. Endo, T. Isshiki, D. Li, and H. Kunieda, "A design method for real-time image denoising circuit using high-level synthesis," in *Proc. 7th Int. Conf. Inf. Commun. Technol. Embedded Syst.*, Bangkok, Thailand, 2016, pp. 30–35.

[56] W. Chen *et al.*, "FPGA-based parallel implementation of SURF algorithm," in *Proc. IEEE 2nd Int. Conf. Parallel Distrib. Syst.*, Wuhan, China, 2016, pp. 308–315.

[57] 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*, Xiamen, China, 2011, pp. 1102–1105.

[58] L. Josipovic, N. George, and P. Ienne, "Enriching C-based high-level synthesis with parallel pattern templates," in *Proc. Int. Conf. Field Program. Technol.*, Xi'an, China, 2016, pp. 177–180.

[59] J. Matai, D. Lee, A. Althoff, and R. Kastner, "Composable, parameterizable templates for high-level synthesis," in *Proc. Design Autom. Test Europe Conf. Exhibit.*, Dresden, Germany, 2016, pp. 744–749.

[60] T. Lu *et al.*, "Memory partitioning-based modulo scheduling for high-level synthesis," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Baltimore, MD, USA, 2017, pp. 1–4.

[61] P. R. Panda *et al.*, "Array scalarization in high level synthesis," in *Proc. 19th Asia South Pac. Design Autom. Conf.*, Singapore, 2014, pp. 622–627.

[62] C. Xiao and E. Casseau, "Improving high-level synthesis effectiveness through custom operator identification," in *Proc. IEEE Int. Symp. Circuits Syst.*, Melbourne, VIC, Australia, 2014, pp. 161–164.

[63] G. Hempel, J. Hoyer, T. Pionteck, and C. Hochberger, "Register allocation for high-level synthesis of hardware accelerators targeting FPGAs," in *Proc. 8th Int. Workshop Reconfigurable Commun. Centric Syst. Chip*, Darmstadt, Germany, 2013, pp. 1–6.

[64] Q. Huang *et al.*, "The effect of compiler optimizations on high-level synthesis for FPGAs," in *Proc. IEEE 21st Annu. Int. Symp. Field Program. Custom Comput. Mach.*, Seattle, WA, USA, 2013, pp. 89–96.

[65] B. C. Schafer, "Process selection for maximum resource sharing in high-level synthesis," in *Proc. Electron. Syst. Level Synth. Conf.*, San Francisco, CA, USA, 2015, pp. 35–40.

[66] J. S. da Silva and S. Bampi, "Area-oriented iterative method for design space exploration with high-level synthesis," in *Proc. IEEE 6th Lat. Amer. Symp. Circuits Syst.*, Montevideo, Uruguay, 2015, pp. 1–4.
[67] D. Liu and B. C. Schafer, "Efficient and reliable high-level synthesis design space explorer for FPGAs," in *Proc. 26th Int. Conf. Field Program. Logic Appl.*, Lausanne, Switzerland, 2016, pp. 1–8.
[68] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of multiple loops on FPGAs using high level synthesis," in *Proc. IEEE 32nd Int. Conf. Comput. Design*, Seoul, South Korea, 2014, pp. 456–463.
[69] C. Lo and P. Chow, "Model-based optimization of high level synthesis directives," in *Proc. 26th Int. Conf. Field Program. Logic Appl.*, Lausanne, Switzerland, 2016, pp. 1–10.
[70] R. Garibotti, B. Reagen, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Using dynamic dependence analysis to improve the quality of high-level synthesis designs," in *Proc. IEEE Int. Symp. Circuits Syst.*, Baltimore, MD, USA, 2017, pp. 1–4.
[71] R. Nane, V. M. Sima, and K. Bertels, "Area constraint propagation in high level synthesis," in *Proc. Int. Conf. Field Program. Technol.*, Seoul, South Korea, 2012, pp. 247–252.
[72] A. Takach, "High-level synthesis: Status, trends, and future directions," *IEEE Des. Test*, vol. 33, no. 3, pp. 116–124, Jun. 2016.

Sakari Lahti (GS'16) received the M.Sc. degrees in engineering physics and computer engineering from the Tampere University of Technology (TUT), Tampere, Finland, in 2002 and 2014, respectively, where he is currently pursuing the Doctoral degree with the Laboratory of Pervasive Computing.

From 2000 to 2002, he was a Research Assistant with the Department of Physics, TUT, and from 2012 to 2014 with the Department of Pervasive Computing, TUT, where he has been a University Teacher with the Laboratory of Pervasive Computing since 2015. His current research interests include high-level synthesis on FPGA, and hardware and system-on-chip designing.

Panu Sjövall (GS'17) received the M.Sc. degree in automation engineering from the Tampere University of Technology (TUT), Tampere, Finland, in 2015, where he is currently pursuing the Doctoral degree with the Laboratory of Pervasive Computing.

He was a Research Assistant with the Department of Pervasive Computing, TUT, from 2014 to 2016, and briefly a Project Researcher with the Department of Pervasive Computing in 2016. Since 2016, he has been with the Laboratory of Pervasive Computing, TUT. His current research interests include hardware and system-on-a-chip designing, high-level synthesis, FPGAs, Linux kernel driver development and video encoding.

Jarno Vanne (M'02) received the M.Sc. degree in information technology and the Ph.D. degree in computing and electrical engineering from the Tampere University of Technology (TUT), Tampere, Finland, in 2002 and 2011, respectively.

He is currently an Assistant Professor with the Laboratory of Pervasive Computing, TUT. He leads the Ultra Video Group that develops open-source Kvazaar HEVC encoder and related multimedia applications on various computing platforms ranging from low-power embedded devices to highly distributed cloud environments. His current research interests include real-time HEVC coding, 3-D/360 video coding for virtual and augmented reality, future video coding standards, intelligent video compression, and high-level synthesis.

Timo D. Hämäläinen (M'95) received the M.Sc. and Ph.D. degrees in electrical engineering from the Tampere University of Technology (TUT), Tampere, Finland, in 1993 and 1997, respectively.

He is currently a Full Professor and the Head of the Laboratory of Pervasive Computing, TUT. He has authored over 70 journals and 240 conference publications. He holds several patents. His current research interests include model-based design methods for multiprocessor systems-on-chip, Kactus2 open source IP-XACT-based system design tool with new visualization paradigms and FPGA accelerated cloud computing for video processing.