This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2015.2513673, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

A Survey and Evaluation of FPGA High-Level Synthesis Tools

Razvan Nane, *Member, IEEE*, Vlad-Mihai Sima, Christian Pilato, *Member, IEEE*, Jongsok Choi, *Student Member, IEEE*, Blair Fort, *Student Member, IEEE*, Andrew Canis, *Student Member, IEEE*, Yu Ting Chen, *Student Member, IEEE*, Hsuan Hsiao, *Student Member, IEEE*, Stephen Brown, *Member, IEEE*, Fabrizio Ferrandi, *Member, IEEE*, Jason Anderson, *Member, IEEE*, Koen Bertels *Member, IEEE*

***Abstract*—High-level synthesis (HLS) is increasingly popular for the design of high-performance and energy-efficient heterogeneous systems, shortening time-to-market and addressing today's system complexity. HLS allows designers to work at a higher-level of abstraction by using a software program to specify the hardware functionality. Additionally, HLS is particularly interesting for designing FPGA circuits, where hardware implementations can be easily refined and replaced in the target device. Recent years have seen much activity in the HLS research community, with a plethora of HLS tool offerings, from both industry and academia. All these tools may have different input languages, perform different internal optimizations, and produce results of different quality, even for the very same input description. Hence, it is challenging to compare their performance and understand which is the best for the hardware to be implemented. We present a comprehensive analysis of recent HLS tools, as well as overview the areas of active interest in the HLS research community. We also present a first-published methodology to evaluate different HLS tools. We use our methodology to compare one commercial and three academic tools on a common set of C benchmarks, aiming at performing an in-depth evaluation in terms of performance and use of resources.**

***Index Terms*—High-Level Synthesis, Survey, Evaluation, Comparison, FPGA, DWARV, LegUp, Bambu.**

I. Introduction

Clock frequency scaling in processors stalled in the middle of the last decade, and in recent years, an alternative approach for high-throughput and energy-efficient processing is based on heterogeneity, where designers integrate software processors and application-specific customized hardware for acceleration, each tailored towards specific tasks [1]. Although specialized hardware has the potential to provide huge acceleration at a fraction of a processor's energy, the main drawback is related to its design. On one hand, describing these components in a hardware description language (HDL) (e.g. VHDL or Verilog) allows the designer to adopt existing tools for RTL and logic synthesis into the target technology. On the other hand, this requires the designer to specify functionality at a low level of abstraction, where cycle-by-cycle behavior is completely specified. The use of such languages requires advanced hardware expertise, besides being cumbersome to develop in. This leads to longer development times that can critically impact the time-to-market.

An interesting solution to realize such heterogeneity and, at the same time, address the time-to-market problem is the combination of reconfigurable hardware architectures, such as field-programmable gate arrays (FPGAs) and high-level synthesis (HLS) tools [2]. FPGAs are integrated circuits that can be configured by the end user to implement digital circuits. Most FPGAs are also reconfigurable, allowing a relatively quick refinement and optimization of a hardware design with no additional manufacturing costs. The designer can modify the HDL description of the components and then use an FPGA vendor toolchain for the synthesis of the bitstream to configure the FPGA. HLS tools start from a high-level *software* programmable language (HLL) (e.g. C, C++, SystemC) to automatically produce a circuit specification in HDL that performs the same function. HLS offers benefits to software engineers, enabling them to reap some of the speed and energy benefits of hardware, without actually having to build up hardware expertise. HLS also offers benefits to hardware engineers, by allowing them to design systems faster at a high-level abstraction and rapidly explore the design space. This is crucial in the design of complex systems [3] and especially suitable for FPGA design where many alternative implementations can be easily generated, deployed onto the target device, and compared. Recent developments in the FPGA industry, such as Microsoft's application of FPGAs in Bing search acceleration [4], and the forthcoming acquisition of Altera by Intel, further underscore the need for using FPGAs as computing platforms with high-level software-amenable design methodologies. HLS has also been recently applied to a variety of applications (e.g. medical imaging, convolutional neural networks, machine learning), with significant benefits in terms of performance and energy consumption [5].

Although HLS tools seem to efficiently mitigate the problem of creating the hardware description, *automatically* generating hardware from software is not easy and a wide range of different approaches have been developed. One approach is to adapt the high-level language to a specific application domain (e.g. dataflow languages for describing streaming applications). These HLS tools can leverage dedicated optimizations or micro-architectural solutions for the specific

R. Nane, V.M. Sima, and K. Bertels are with Delft University of Technology, The Netherlands.

C. Pilato was with Politecnico di Milano, Italy, and is now with Columbia University, NY, USA.

F. Ferrandi is with Politecnico di Milano, Italy.

J. Choi, B. Fort, A. Canis, Y.T. Chen, H. Hsiao, S. Brown, and J. Anderson are with University of Toronto, Canada.

domain. However, the algorithm designer, who is usually a software engineer, has to understand how to properly update the code. This approach is usually time consuming and error prone. For this reason, some HLS tools offer complete support for a standard high-level language, such as C, giving complete freedom to the algorithm designer. Understanding the current HLS research directions, the different HLS tools available and their capabilities is a difficult challenge, and a thoughtful analysis is lacking in the literature to cover all these aspects. For example, [6] was a small survey of existing HLS tools with a static comparison (on criteria such as the documentation available or the learning curve) of their features and user experience. However, the tools have not been applied to benchmarks, nor were the results produced by the tools compared. Indeed, given an application to be implemented as a hardware accelerator, it is crucial to understand which HLS tool better fits the characteristics of the algorithm. For this reason, we believe that it is important to have a comprehensive survey of recent HLS tools, research directions, as well as a systematic method to evaluate different tools on the same benchmarks in order to analyze the results.

In this paper, we present a thorough analysis of HLS tools, current HLS research thrusts, as well as a detailed way to evaluate different state-of-the-art tools (both academic and commercial) on performance and resource usage. The three academic tools considered are DWARV [7], BAMBU [8] and LEGUP [9] – tools under active development in three institutions and whose developers are co-authoring this paper. The contributions of this work are:

- A thorough evaluation of past and present HLS tools (Section II).
- A description of HLS optimizations and problems wherein active research is underway (Section III).
- The first-published comprehensive in-depth evaluation (Section IV) and discussion (Section V) of selected commercial and academic HLS tools in terms of performance and area metrics.

This analysis shows that industry and academia are closely progressing together towards efficient methods to automatically design application-specific customized hardware accelerators. However, several research challenges remain open.

II. OVERVIEW OF HIGH-LEVEL SYNTHESIS TOOLS

In this section, we present an overview of academic and commercial HLS tools. The presentation will be done according to a classification of the design input language as shown in Fig. 1. We distinguish between two major categories, namely tools that accept domain-specific languages (DSLs) and tools that are based on general-purpose programmable languages (GPLs). DSLs are split into *new languages* invented specially for a particular tool-flow and GPL-based dialects, which are languages based on a GPL (e.g. C) extended with specific constructs to convey specific hardware information to the tool. Under each category, the corresponding tools are listed in green, red or blue fonts, where green represents the tool being *in use*, red implies the tool is *abandoned* and blue implies *N/A*, meaning that no information is currently known about its status. Furthermore, the bullet type, defined in the figure's legend, denotes the target application domain for which the tool can be used. Finally, tool names which are underlined in the figure mean that the tool also supports SystemC as input. Using DSLs or SystemC raises challenges for adoption of HLS by software developers. In this section, due to space limitations, we describe only the unique features of each tool. For general information (e.g. target application domain, support for floating/fixed-point arithmetic, automatic testbench generation), the reader is referred to Table I. We first introduce the academic HLS tools evaluated in this study, before moving onto highlight features of other HLS tools available in the community (either commercial or academic).

A. Academic HLS Tools Evaluated in This Study

DWARV [7] is an academic HLS compiler developed at Delft University of Technology. The tool is based on the CoSy commercial compiler infrastructure [10] developed by ACE. The characteristics of DWARV are directly related to the advantages of using CoSy, which are its modular and robust back-end, and that is easily extensible with new optimizations.

BAMBU [8] is an academic HLS tool developed at Politecnico di Milano and first released in 2012. BAMBU is able to generate different pareto-optimal implementations to trade-off latency and resource requirements, and to support hardware/software partitioning for designing complex heterogeneous platforms. Its modular organization allows the evaluation of new algorithms, architectures, or synthesis methods. BAMBU leverages the GCC compiler's many compiler-based optimizations and implements a novel memory architecture to efficiently support complex constructs of the C language (e.g., function calls, pointers, multi-dimensional arrays, structs) [11]. It is also able to support different data types, including floating-point arithmetic, in order to generate optimized micro-architectures.

LEGUP [9] is a research compiler developed at the University of Toronto, first released in 2011 and currently on its forth public release. It accepts a C-language program as input and operates in one of two ways: 1) it synthesizes the entire C program to hardware, or 2) it synthesizes the program to a hybrid system comprising a processor (a MIPS soft processor or ARM) and one or more hardware accelerators. In the latter flow, the user designates which C functions to implement as accelerators. Communication between the MIPS/ARM and accelerators is through Altera's memory-mapped on-chip bus interface (LEGUP is designed specifically to target a variety of Altera FPGA families). For hardware synthesis, most of the C language is supported, with the exception of dynamic memory allocation and recursion. LEGUP is built within the open-source LLVM compiler framework [12]. Compared to other HLS tools, LEGUP has several unique features. It supports Pthreads and OpenMP, where parallel software threads are automatically synthesized into parallel-operating hardware. Automated bitwidth reduction can be invoked to shrink datapath widths based on compile-time (static) variable range and bitmask analysis. Multi-cycle path analysis and register removal are also supported, wherein LEGUP eliminates

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2015.2513673, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

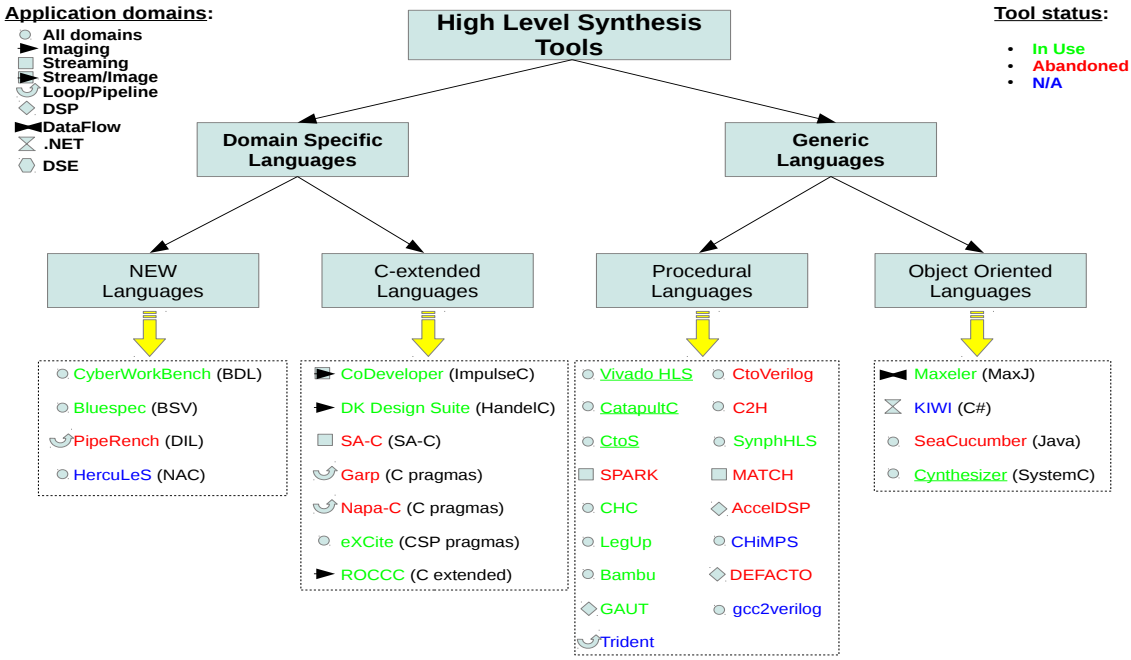

Fig. 1: Classification of High-Level Synthesis Tools Based on the Input Language.

TABLE I: Overview of High-Level Synthesis Tools.

Status	Compiler	Owner	License	Input	Output	Year	Domain	TestBench	FP	FixP
In Use	eXCite	Y Explorations	Commercial	C	VHDL/Verilog	2001	All	Yes	No	Yes
	CoDeve-loper	Impulse Accelerated	Commercial	Impulse-C	VHDL Verilog	2003	Image Streaming	Yes	Yes	No
	Catapult-C	Calypto Design Systems	Commercial	C/C++ SystemC	VHDL/Verilog SystemC	2004	All	Yes	No	Yes
	Cynthesizer	FORTE	Commercial	SystemC	Verilog	2004	All	Yes	Yes	Yes
	Bluespec	BlueSpec Inc.	Commercial	BSV	SystemVerilog	2007	All	No	No	No
	CHC	Altium	Commercial	C subset	VHDL/Verilog	2008	All	No	Yes	Yes
	CtoS	Cadence	Commercial	SystemC TLM/C++	Verilog SystemC	2008	All	Only cycle accurate	No	Yes
	DK Design Suite	Mentor Graphics	Commercial	Handel-C	VHDL Verilog	2009	Streaming	No	No	Yes
	GAUT	U. Bretagne	Academic	C/C++	VHDL	2010	DSP	Yes	No	Yes
	MaxCompiler	Maxeler	Commercial	MaxJ	RTL	2010	DataFlow	No	Yes	No
	ROCCC	Jacquard Comp.	Commercial	C subset	VHDL	2010	Streaming	No	Yes	No
	Synphony C	Synopsys	Commercial	C/C++	VHDL/Verilog SystemC	2010	All	Yes	No	Yes
	Cyber-WorkBench	NEC	Commercial	BDL	VHDL Verilog	2011	All	Cycle/ Formal	Yes	Yes
	LegUp	U. Toronto	Academic	C	Verilog	2011	All	Yes	Yes	No
	Bambu	PoliMi	Academic	C	Verilog	2012	All	Yes	Yes	No
	DWARV	TU. Delft	Academic	C subset	VHDL	2012	All	Yes	Yes	Yes
	VivadoHLS	Xilinx	Commercial	C/C++ SystemC	VHDL/Verilog SystemC	2013	All	Yes	Yes	Yes
N/A	Trident	Los Alamos NL	Academic	C subset	VHDL	2007	Scientific	No	Yes	No
	CHiMPS	U. Washington	Academic	C	VHDL	2008	All	No	No	No
	Kiwi	U. Cambridge	Academic	C#	Verilog	2008	.NET	No	No	No
	gcc2verilog [45]	U. Korea	Academic	C	Verilog	2011	All	No	No	No
	HercuLeS	Ajax Compiler	Commercial	C/NAC	VHDL	2012	All	Yes	Yes	Yes
Abandoned	Napa-C	Sarnoff Corp.	Academic	C subset	VHDL/Verilog	1998	Loop	No	No	No
	DEFACTO	U. South Cailf.	Academic	C	RTL	1999	DSE	No	No	No
	Garp	U. Berkeley	Academic	C subset	bitstream	2000	Loop	No	No	No
	MATCH	U. Northwest	Academic	MATLAB	VHDL	2000	Image	No	No	No
	PipeRench	U.Carnegie M.	Academic	DIL	bitstream	2000	Stream	No	No	No
	SeaCucumber	U. Brigham Y.	Academic	Java	EDIF	2002	All	No	Yes	Yes
	SA-C	U. Colorado	Academic	SA-C	VHDL	2003	Image	No	No	No
	SPARK	U. Cal. Irvine	Academic	C	VHDL	2003	Control	No	No	No
	AccelDSP	Xilinx	Commercial	MATLAB	VHDL/Verilog	2006	DSP	Yes	Yes	Yes
	C2H	Altera	Commercial	C	VHDL/Verilog	2006	All	No	No	No
	CtoVerilog	U. Haifa	Academic	C	Verilog	2008	All	No	No	No

registers on some paths permitted more than a single cycle to complete, generating constraints for the back-end of the toolflow accordingly.

B. Other HLS Tools

CyberWorkBench (CWB) [13] is a set of synthesis, verification and simulation tools intended for system-level design. The tool input is the Behavioral Description Language (BDL), i.e. a superset of the C language extended with constructs to express hardware concepts. Examples of such constructs are user-defined bitwidth for variables, synchronization, explicit clock boundaries specification, and concurrency.

Bluespec Compiler (BSC) [14] is a tool that uses Bluespec System Verilog (BSV) as the design language. BSV is a high-level functional HDL based on Verilog and inspired by Haskell, where modules are implemented as a set of rules using Verilog syntax. The rules are called *Guarded Atomic Actions* and express behaviour in the form of concurrently cooperating FSMs [15]. Using this language, and implicitly the BSC tool, requires developers with specific expertise.

PipeRench [16] was an early reconfigurable computing project. The PipeRench compiler was intended solely for generating reconfigurable pipelines in stream-based media applications. The source language is a *dataflow intermediate language*, which is a single-assignment language with C operators. The output of the tool is a bitstream to configure the reconfigurable pipeline in the target circuit.

HercuLeS [17] is a compiler that uses an NAC (N-address code) IR (intermediate representation), which is a new typed-assembly language created by a front-end available through GCC Gimple. The work deals only with complete applications targeting FPGAs.

CoDeveloper [18] is the HLS design environment provided by Impulse Accelerated Technologies. Impulse-C is based on a C-language subset to which it adds communicating sequential processes (CSP)-style extensions. These extensions are required for parallel programming of mixed processor and FPGA platforms. Because the basic principle of the CSP programming model consists of processes that have to be independently synchronized and streams for inter-process communication, the application domain is limited primarily to image processing and streaming applications.

DK Design Suite [19] uses Handel-C as the design language, which is based on a rich subset of the C language extended with hardware-specific language constructs. The user however needs to specify timing requirements, and to describe the parallelization and synchronization segments in the code explicitly. In addition, the data mapping to different memories has to be manually performed. Because of these language additions, the user needs advanced hardware knowledge.

Single-Assignment C (SA-C) [20] is a compiler that uses a C language variant in which variables can be set only once. This work provided the inspiration for the later ROCCC compiler. The language introduces new syntactical constructs that require application rewriting.

The *Garp* [21] project's main goal was to accelerate loops of general-purpose software applications. It accepts C as input and generates hardware code for the loop.

The *Napa-C* [22] project was one of the first to consider high-level compilation for systems which contain both a microprocessor and reconfigurable logic. The NAPA C compiler, implemented in SUIF and targeting National Semiconductor's NAPA1000 chip, performed semantic analysis of the pragma-annotated program and co-synthesized a conventional program executable for the processor, and a configuration bit stream.

In *eXCite* [23], communication channels have to be inserted manually to describe the communication between the software and hardware. These channels can be streaming, blocking or indexed (e.g. for handling arrays). Different types of communication between the software and hardware parts (e.g. streaming, shared memory) are possible.

The *ROCCC* [24] project focused mainly on the parallelization of heavy-compute-density applications having little control. This restricts its application domain to streaming applications, and it means that the input C is limited to a subset of the C language. For example, only perfectly nested loops with fixed stride, operating on integer arrays are allowed.

Catapult-C [25] is a commercial high-level synthesis tool initially oriented towards the ASIC hardware developer, however, it now targets both FPGAs and ASICs. It offers flexibility in choosing the target technology, external libraries, setting the design clock frequency, mapping function parameters to either register, RAM, ROM or streaming interfaces.

C-to-Silicon (CtoS) [26], offered by Cadence, offers support for both control- and dataflow applications. Since it accepts SystemC as input, it is possible to accurately specify different interface types, from simple function array parameters to cycle-accurate transmission protocols.

SPARK [27] was targeted to multimedia and image processing applications along with control-intensive microprocessor functional blocks. The compiler generated synthesizable VHDL that could be mapped to both ASICs or FPGAs.

The *C to Hardware Compiler* [28] generates hardware to be offloaded onto a *Application Specific Processor (ASP)* core, for which verification has to be done manually by loading and executing the generated design on an Altium Desktop NanoBoard NB2DSK01.

A distinct feature of the *GAUT* [29] project is that besides the processing accelerator, it can generate both communication and memory units. A testbench is also automatically generated to apply stimuli to the design and to analyze the results for validation purposes. Fixed-point arithmetic is supported through Mentor Graphics Algorithmic C class library.

Trident [30] is a compiler that is an offshoot of an earlier project called Sea Cucumber [31]. It generates VHDL-based accelerators for scientific applications operating on floating-point data starting from a C-language program. Its strength is in allowing users to select floating-point operators from a variety of standard libraries, such as *FPLibrary* and *Quixilica*, or to import their own.

C2H [32] was an HLS tool offered by Altera Corporation since 2006. The tool is technology dependent, generating accelerators that can only communicate via an Altera Avalon bus with an Altera NIOS II configurable soft processor. Furthermore, using this tool required advanced hardware design knowledge in order to configure and connect the accelerators

to the rest of the system – tasks performed in Altera's development environment.

Synphony C [33], formerly PICO [34], is an HLS tool for hardware DSP design offered by Synopsys. The tool can support both streaming and memory interfaces and allows for performance-related optimizations to be fine-tuned (e.g. loop unrolling, loop pipelining). Floating point operations are not permitted, but the programmer can use fixed-point arithmetic. Comparison results published by BDTi [35] showed that performance and area metrics for Synphony-produced circuits are comparable with those obtained with AutoESL (the product that become Vivado HLS when acquired by Xilinx).

The goal of the *MATCH* [36] software system was to translate and map MATLAB code to heterogeneous computing platforms for signal and image processing applications. The MATCH technology was later transferred to a startup company, AccelChip [37], bought in 2006 by Xilinx but discontinued in 2010. The tool was one of the few on the market that started from a MATLAB input description to generate VHDL or Verilog. Key features of the product were automation conversion of floating point to fixed point.

The *CHiMPS* compiler [38] targets applications for high-performance. The distinctive feature of CHiMPS is its *many-cache*, which is a hardware model that adapts the hundreds of small, independent FPGA memories to the specific memory needs of an application. This allows for many simultaneous memory operations per clock cycle to boost performance.

DEFACTO [39] is one of the early design environments that proposed hardware/software co-design solutions as an answer to increasing demands for computational power. DEFACTO is composed of a series of tools such as a profiler, partitioner and software and hardware compilers to perform fast design space exploration given a set of design constraints.

MaxCompiler [40] is a data-flow specific HLS tool. The compiler accepts MaxJ, a Java-based language, as input and generates synthesizable code for the hardware data-flow engines provided by Maxeler's hardware platform.

The *Kiwi* [41] programming library and its associated synthesis system generates FPGA co-processors (in Verilog) from C# programs. Kiwi allows the programmer to use parallel constructs such as events, monitors and threads, which are closer to hardware concepts than classical software constructs.

Sea Cucumber [31] is a Java-based compiler that generates EDIF netlists and adopts the standard Java thread model, augmented with a communication model based on CSP.

Cynthesizer [42], recently acquired by Cadence, includes formal verification between RTL and gates, power analysis, and several optimizations, such as support for floating-point operations with IEEE-754 single/double precision.

Vivado HLS [43], formerly AutoPilot [44], was developed initially by AutoESL until it was acquired by Xilinx in 2011. The new improved product, which is based on LLVM, was released early 2013, and includes a complete design environment with abundant features to fine-tune the generation process from HLL to HDL. C, C++ and SystemC are accepted as input, and hardware modules are generated in VHDL, Verilog and SystemC. During the compilation process, it is possible to apply different optimizations, such as operation chaining, loop pipelining, and loop unrolling. Furthermore, different parameter mappings to memory can be specified. Streaming or shared memory type interfaces are both supported to simplify accelerator integration.

III. High-Level Synthesis (HLS) Optimizations

HLS tools feature several optimizations to improve the performance of the accelerators. Some of them are borrowed from the compiler community, while others are specific for hardware design. In this section, we discuss these HLS optimizations, which are also current research trends for the HLS community.

A. Operation Chaining

Operation chaining is an optimization that performs operation scheduling within the target clock period. This requires the designer to "chain" two combinational operators together in a single cycle in a way that *false paths* are avoided [46]. Concretely, if two operations are dependent in the data-flow graph and they can both complete execution in a time smaller than the target clock period, then they can be scheduled in the same cycle; otherwise at least two cycles are needed to finish execution, along with a register for the intermediate result. Generally, chaining reduces the number of cycles in the schedule, improving performance and reducing the global number of registers in the circuit. However, this is highly technology dependent and requires an accurate characterization of the resource library (see Section III-E).

B. Bitwidth Analysis and Optimization

Bitwidth optimization is a transformation that aims to reduce the number of bits required by datapath operators. This is a very important optimization because it impacts all non-functional requirements (e.g. performance, area, power) of a design, without affecting its behavior. Differently from GPP compilers, which are designed to target a processor with a fixed-sized datapath (usually 32 or 64 bits), a hardware compiler can exploit specialization by generating custom-size operators (i.e. functional units) and registers. As a direct consequence, we can select the minimal number of bits required for an operation and/or storage of the specific algorithm, which in turns leads to minimal space used for registers, smaller functional units that translate into less area, less power, and shorter critical paths. However, this analysis cannot be usually completely automated since it often requires specific knowledge of the algorithm and the input datasets.

C. Memory Space Allocation

FPGAs contain multiple memory banks in the form of distributed block RAMs (BRAMs) across the device. This allows the designer to partition and map software data structures onto dedicated BRAMs in order to implement fast memory accesses at low cost. As a result, the scheduler can perform multiple memory operations in one cycle once it is able to statically determine that they access different memories in the same cycle without contention. This feature is similar to the allocation of different memory spaces used in the

embedded systems domain. Using multiple BRAMs increases the available parallelism. On the other hand, these memory elements have a limited number of memory ports and the customization of memory accesses may require the creation of an efficient multi-bank architecture to avoid limiting the performance [47].

D. Loop Optimizations

Hardware acceleration is particularly important for algorithms with compute-intensive loops. *Loop pipelining* is a key performance optimization for loops implemented in hardware. This optimization exploits loop-level parallelism by allowing a loop iteration to start before the completion of its predecessor, provided that data dependences are satisfied. The concept is related to software pipelining [48], which has widely been applied in VLIW processors. A key concept in loop pipelining is the *initiation interval* (II), which is the number of clock cycles between successive loop iterations. For high throughput, it is desirable that II be as small as possible, ideally one, which implies that a new loop iteration is started every cycle. Achieving the minimum II for a given design can be impeded by two factors: 1) resource constraints, and 2) loop-carried dependencies. Regarding resource constraints, consider a scenario where a loop body contains 3 load operations, and 1 store operation. In this case, achieving an II less than two is impossible if the memory has two ports, since each loop iteration has 4 memory operations. For this reason, loop optimizations are frequently combined with multi-bank architecture to fully exploit the parallelism [47]. With respect to loop-carried dependencies, if a loop iteration depends on a result computed in a prior iteration, that data dependency may restrict the ability to reduce II, as it may be necessary to delay commencing an iteration until its dependent data has been computed.

For example, DWARV leverages CoSy to implement loop pipelining. The heuristic applied is based on swing modulo scheduling [49], which considers operation latencies between loop instructions to move conflicting instructions and reduce the II. However, due to the high availability of resources in FPGAs, the loop pipelining algorithm for hardware generation can be relaxed. This can be accomplished by fixing the II to a desired value, i.e. based on a required design throughput, and then generating enough hardware (e.g. registers) to accommodate the particular II.

Recent research has focused on loop pipelining for nested loops. Consider, for example, a doubly-nested loop whose outermost loop (with induction variable i) iterates 100 times, and whose innermost one (with induction variable j) iterates up to i times. The iteration space traversed by i and j can be viewed as a polyhedron (in this case, a triangle) and analytically analyzed with the *polyhedral model* [50]. Applying loop transformations (e.g. exchanging the outer and inner loop) result in different polyhedra, and potentially different IIs. Polyhedral-based optimizations have been applied to synthesize memory architectures [51], improve throughput [52], and optimize resource usage [53].

E. Hardware Resource library

In the process of HLS, in order to generate an efficient implementation that meets timing requirements while minimizing the use of resources, it is essential to determine how to implement each operation. Specifically, the front-end phase first inspects the given behavioral specification and identifies operations characteristics, such as the type of each operation (e.g. arithmetic or non-arithmetic), its operand types (e.g. integer, float), and its bit-width. At this stage, some operations may benefit from specific optimizations. For example, multiplications or divisions by a constant are typically transformed into operations that use only shifts and adds [54], [55] in order to improve area and timing. All these characteristics are then used during the *module allocation* phase, where the resulting operations are associated with functional units contained in the resource library [46]. This heavily impacts the use of resources and the timing of the resulting circuit. Hence, the proper composition of such a library and its characterization is crucial for efficient HLS.

The library of functional units can be quite rich and may contain several implementations for each single operation. On one hand, the library usually includes resources that are specific for the technology provider (e.g. the FPGA vendor). Some of these resources may leverage vendor-specific intrinsics or IP generators. In this case the module allocation will exploit resources that have been explicitly tailored and optimized for the specific target. This is usually adopted by HLS tools that are specific for some FPGA vendors (e.g. [43]). The library may also contain resources that are expressed as templates in a standard hardware description language (i.e. Verilog or VHDL). These templates can be retargeted and customized based on characteristics of the target technology, like in FloPoCo [56]. In this case, the underlying logic synthesis tool can determine which is the best architecture to implement each function. For example, multipliers can be mapped either on dedicated DSP blocks or implemented with LUTs.

To perform aggressive optimizations, each component of the library needs to be annotated with information useful during the entire HLS process, such as resource occupation and latency for executing the operations. There are several approaches for library characterization. The first approach performs a rapid logic synthesis during the scheduling and binding of the operations to determine the most suitable candidate resources, like in Cadence's C-to-Silicon [57]. However, this approach has a high cost in terms of computation time, especially when the HLS is repeatedly performed for the same target. An alternative approach is to pre-characterize all resources in advance, as done in BAMBU [8]. The performance estimation starts with a generic template of the functional unit, which can be parametric with respect to bitwidths and pipeline stages. Latency and resource occupation are then obtained by synthesizing each configuration and storing the results in the library. Mathematical models can be built on top of these actual synthesis values [58], [59]. Additionally, this information can also be coupled with delays obtained after the place-and-route phase. This may improve the maximum frequency and the design latency and it makes the HLS results

more predictable [60].

F. Speculation and Code Motion

Most HLS scheduling techniques can extract parallelism only within the same control region (i.e. the same CDFG basic block). This can limit the performance of the resulting accelerator, especially in control-intensive designs. *Speculation* is a code-motion technique that allows operations to be moved along their execution traces, possibly anticipating them before the conditional constructs that control their execution [61], [62], [63], [64]. A software compiler is less likely to use this technique since, in a sequential machine, they may delay the overall execution with computations that are unnecessary in certain cases. In hardware, however, speculated operations can often be executed in parallel with the rest of the operations. Their results simply will be simply maintained or discarded based on later-computed branch outcomes.

G. Exploiting Spatial Parallelism

A primary mechanism through which hardware may provide higher speed than a software implementation is by instantiating multiple hardware units that execute concurrently (*spatial parallelism*). HLS tools can extract fine-grained instruction-level parallelism by analyzing data dependencies and loop-level parallelism via loop pipelining. It is nevertheless difficult to automatically extract large amounts of coarse-grained parallelism, as the challenges therein are akin to those faced by an auto-parallelizing software compiler. A question that arises, therefore, is how to specify hardware parallelism to an HLS tool whose input is a software programming language. With many HLS tools, a designer synthesizes an accelerator and then manually writes RTL that instantiates multiple instances of the synthesized core, steering input/output data to/from each, accordingly. However, this approach is error prone and requires hardware expertise. An alternative approach is to support the synthesis of software parallelization paradigms.

LEGUP supports the HLS of pthreads and OpenMP [65], which are two standard ways of expressing parallelism in `C` programs, widely used by software engineers. The general idea is to synthesize parallel software threads into an equal number of parallel-operating hardware units. With pthreads, a user can express both task and data-level spatial parallelism. In the former, each hardware unit may be performing a different function, and in the latter, multiple hardware units perform the same function on different portions of an input dataset. LEGUP also supports the synthesis of two standard pthreads synchronization constructions: mutexes and barriers. With OpenMP, the authors have focused on supporting the aspects of the standard that target loop parallelization, e.g. an N-iteration loop with no loop-carried dependencies can be split into pieces that are executed in parallel by concurrently operating hardware units. An interesting aspect of LEGUP is the support for *nested parallelism*: threads forking threads. Here, the threads initially forked within a program may themselves fork other threads, or contain OpenMP parallelization constructs. A limitation of the LEGUP work is that the number of parallel hardware units instantiated must exactly match the number of software threads forked since in hardware there is no support for context switching.

Altera has taken an different approach with their OpenCL SDK [66], which supports HLS of OpenCL programs. The OpenCL language is a variant of `C` and is heavily used for parallel programming of graphics processing units (GPUs). With OpenCL, one generally launches hundreds or thousands of threads that are relatively fine-grained, for example, each computing a vector dot product, or even an individual scalar multiplication. Altera synthesizes OpenCL into a deeply pipelined FPGA circuit that connects to an x86-based host processor over PCIe. The support for OpenCL HLS allows Altera to compete directly with GPU vendors, who have been gaining traction in the high-performance computing market.

H. If-Conversion

If-conversion [67] is a well-known software transformation that enables predicated execution, i.e. instructions are executed only when its predicate or guard evaluates to true. The main objective of this transformation is to schedule in parallel instructions from disjoint execution paths created by selective statements (e.g. `if` statements). The goals are two fold. First, it increases the number of parallel operations. Second, it facilitates pipelining by removing control dependencies within the loop, which may shorten the loop body schedule. In software, this leads to a 34% performance improvement, on average [68]. However, if-conversion should be enabled only when the branches have a balanced number of cycles required to complete execution. When this is not the case, predicated execution incurs a slowdown in execution time because, if the shorter branch is taken, useless instructions belonging to the longer unselected branch will need to be checked before executing useful instructions can be resumed. Therefore, different algorithms have been proposed to decide when it is beneficial to apply if-conversion and when the typical conditional jump approach should be followed. For example, in [69], a generic model to select the fastest implementation for if-then-else statements is proposed. This selection is done according to the number of implicated if-statements, as well as the balance characteristics. The approach for selecting if-conversion on a case-by-case basis changes for hardware compilers generating an FPGA hardware circuit. This is because resources can be allocated as-needed (subject to area or power constraints), and therefore, we can schedule branches in a manner that does not affect the branch-minimal schedule. Data and control instructions can be executed in parallel and we can insert "jumps" to the end of the if-statement to short-cut the execution of (useless) longer branch instructions when a shorter path is taken. This was demonstrated in [70] (incorporated in DWARV), which proposed a lightweight if-conversion scheme adapted for hardware generation. Furthermore, the work showed that such a lightweight predicative scheme is beneficial for hardware compilers, with performance always at least as good as when no if-conversion is enabled.

IV. Evaluation of High-Level Synthesis Tools

In this section, we define a common environment to evaluate four HLS tools, i.e. one commercial and three academic:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2015.2513673, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

DWARV, BAMBU, and LEGUP. With LEGUP we target the fastest speedgrade of the Altera Stratix V family [71], while with the other tools we target the fastest speedgrade of the Xilinx Virtex-7 family [72]. Both Stratix V and Virtex-7 are 28nm state-of-the-art high-performance FPGAs fabricated by TSMC. The primary combinational logic element in both architectures is a dual-output 6-input look-up-table (LUT).

Three metrics are used to evaluate circuit performance: *maximum frequency* (*FMax* in MHz), *cycle latency* (i.e. the number of clock cycles needed for a benchmark to complete the computation), and *wall-clock time* (minimum clock period $\times$ cycle latency). Clock period (and the corresponding *FMax*) is extracted from post-routing static timing analysis. Cycle latency is evaluated by simulating the resulting RTL circuits using ModelSim (discussed further below). We do not include in the evaluations the HLS tool execution times as this time is negligible in comparison with the synthesis, mapping, placement and routing time.

To evaluate area, we consider logic, DSP and memory usage. For logic area, in Xilinx devices, we report the total number of fracturable 6-LUTs, each of which can be used to implement any single function of up to 6 variables, or any two functions that together use at most 5 distinct variables. For Altera, we report the total number of used adaptive logic modules (ALMs), each of which contains one fracturable 6-LUT, that can implement any single function of up to 6 variables, any two 4-variable functions, a 5- and 3-variable function, and several other dual-function combinations. With respect to DSP usage, we consider the DSP units in Altera and Xilinx devices to be roughly equivalent (Xilinx devices contain hardened 25×18 multipliers, whereas Altera devices contain hardened 18×18 multipliers). For memory, we report the total number of dedicated blocks used (e.g. BRAMs), which are equivalent to 18Kb in Virtex-7 and 20Kb in Stratix V.

A. Potential Sources of Inaccuracy

Although we have endeavored to make the comparison between tools as fair as possible, we discuss potential sources of inaccuracy to better understand the results. First, the evaluated HLS tools are built within different compilers (e.g. BAMBU is built within GCC, LEGUP within LLVM, and DWARV within CoSy), and target different FPGA devices. It is thus impossible to perfectly isolate variations in circuit area and speed attributable to the HLS tools versus other criteria. Each compiler framework has a different set of optimizations that execute before HLS, with potentially considerable impact on HLS results. Likewise, we expect that Altera's RTL and logic synthesis, placement and routing, are different than those within Xilinx's tool. Moreover, while the chosen Virtex-7 and Stratix V are fabricated in the same TSMC process, there are differences in the FPGA architecture itself. For example, as mentioned above, the fracturable 6-LUTs in Altera FPGAs are more flexible than the fracturable 6-LUTs in Xilinx FPGAs, owing to the Altera ALMs having more inputs. This may vary the final resource requirements for the accelerators. Finally, although we have selected the fastest speedgrade for each vendor's device, we cannot be sure whether the fraction of die binned in Xilinx's fastest speedgrade is the same as that for Altera because this information is kept proprietary by the vendors.

TABLE II: Benchmark Characteristics & Target Frequencies for Optimized Flow (MHz).

Benchmark	Domain	Source	BAMBU	DWARV	LEGUP	Commercial
adpcm_encode	Comm	CHStone	333	150	333	400
aes_encrypt	Encrypt	CHStone	250	200	333	363
aes_decrypt	Encrypt	CHStone	250	200	1000	312
gsm	Comm	CHStone	200	150	333	400
sha	Encrypt	CHStone	200	200	333	400
blowfish	Encrypt	CHStone	250	200	200	400
dfadd	Arith	CHStone	250	200	333	400
dfdiv	Arith	CHStone	250	150	200	400
dfmul	Arith	CHStone	250	150	200	400
dfsin	Arith	CHStone	250	100	200	303
jpeg	Media	CHStone	250	N/A	1000	400
mips	Compute	CHStone	400	300	200	400
motion	Media	CHStone	250	150	200	ERR
satd	Compute	DWARV	455	100	100	400
sobel	Media	DWARV	500	300	1000	285
bellmanford	Compute	DWARV	500	300	333	400
matrix	Arith	Bambu	250	300	250	400

Other differences relate to tool assumptions, e.g. about memory implementation. For each benchmark kernel, some data is kept local to the kernel (i.e. in BRAMs instantiated within the module), whereas other data is considered "global", kept outside the kernel and accessed via a memory controller. As an example, in LEGUP, a data is considered local when, at compile time, is proven to solely be accessed within the kernel (e.g. an array declared within the kernel itself and used as a scratch pad). The various tools evaluated do not necessarily make the same decisions regarding which data is kept local versus global. The performance and area numbers reported reflect the kernel itself and do not include the global memory. The rationale behind this decision is to focus on the results on the HLS-generated portion of the circuit, rather than on the integration with the rest of the system.

B. Benchmark Overview

The synthesized benchmark kernels are listed in Table II, where we mention in the second and third columns the application domain of the corresponding kernel, as well as its source. Most of the kernels have been extracted from the C-language CHStone benchmark suite [73], with the remainder being from DWARV and BAMBU. The selected functions originate from different application domains, which are control-flow, as well as data-flow dominated as we aim at evaluating generic (non-application-specific) HLS tools.

An important aspect of the benchmarks used in this study is that input and golden output vectors are available for each program. Hence, it is possible to "execute" each benchmark with the built-in input vectors, both in software and also in HLS-generated RTL using ModelSim. The RTL simulation permits extraction of the total cycle count, as well as enables functional correctness checking.

C. HLS Evaluation

We performed two sets of experiments to evaluate the compilers. In the first experiment, we executed each tool in a

TABLE III: Optimizations Used (Letter in () Refers to Subsection in Section III). v: USED; x: UNUSED.

HLS Tool	Compiler Framework	Target FPGA	OC(A)	BA(B)	MS(C)	LO(D)	HwRL(E)	Sp(F)	SP(G)	IC(H)
LEGUP	LLVM	Altera	v	v	v	v	x	x	v	v
BAMBU	GCC	Xilinx	v	v	v	x	v	v	x	v
DWARV	CoSy	Xilinx	v	v	v	v	x	x	x	v
Commercial	Unknown	Xilinx	v	v	v	v	v	v	v	v

"push-button" manner using all of its default settings, which we refer to as *standard-optimization*. The first experiment thus represents what a user would see running the HLS tools "out of the box". We used the following default target frequencies: 250 MHz for BAMBU, 150 MHz for DWARV, and 200 MHz for LEGUP. For the commercial tool, we decided to use a default frequency of 400 MHz. In the second experiment, we manually optimized the programs and constraints for the specific tools (by using compiler flags and code annotations to enable various optimizations) to generate *performance-optimized* implementations. Table III lists for each tool the optimizations enabled in this second experiment. As we do not have access to the source of the commercial tool, its list is based on the observations done on the available options and on the inspection of the generated code. The last four columns of Table II show the HLS target frequencies used for the optimized experiment. It should be noted that there is no strict correlation between these and the actual post place and route frequencies obtained after implementing the designs (shown in tables IV and V) due to the actual vendor-provided back-end tools that perform the actual mapping, placing and routing steps. This is explained by the inherently approximate timing models for computing in HLS. The target frequency used as input to the HLS tools should be regarded only as an indication of how much operation chaining can be performed. As a rule of thumb, in order to implement a design at some frequency, one should target a higher frequency in HLS.

Table IV shows performance metrics (e.g. number of cycles, maximum frequency after place & route, and wall-clock time) obtained in the standard-optimization scenario, while Table V shows the same performance metrics obtained in the performance-optimized scenario. The *ERR* entries denote errors that prevented us from obtaining complete results for the corresponding benchmarks (e.g. compiler segmentation error). Observe that geometric mean data is included at the bottom of the rows. Two rows of geomean are shown: the first includes only those benchmarks for which all tools were successful; the second includes *all* benchmarks, and is shown for BAMBU and LEGUP. In the *standard-optimization* results in Table IV, we see that the commercial tool is able to achieve the highest $Fmax$; BAMBU implementations have the lowest cycle latencies; and BAMBU and LEGUP deliver roughly the same (and lowest) average wall-clock time. However, we also observe that no single tool delivers superior results for *all* benchmarks. For example, while DWARV does not provide the lowest wall-clock time on average, it produced the best results (among the academic tools) for several benchmarks, including *aes_decrypt* and *bellmanford*.

For the performance-optimized results in Table V, a key takeaway is that performance is drastically improved when the constraints and source code input to the HLS tools are tuned. For the commercial tool, geomean wall-clock time is reduced from 37.1 to 19.9μs (1.9×) in the optimized results. For BAMBU, DWARV and LEGUP, the wall-clock time reductions in the optimized flow are 1.6×, 1.7× and 2×, respectively, on average (comparing values in the GEOMEAN row of the table). It is interesting that, for all the tools, the average performance improvements in the optimized flow were roughly the same. From this, we conclude that one can expect ∼1.6-2× performance improvement, on average, from tuning code and constraints provided to HLS. We also observe that, from the performance angle, the academic tools are comparable to the commercial tool. BAMBU and LEGUP, in particular, deliver superior wall-clock time to commercial, on average.

For completeness, the area-related metrics are shown in tables VI and VII for the standard and optimized flows, respectively. Comparisons between LEGUP and the other tools are more difficult in this case, owing to architectural differences between Stratix V and Virtex-7. Among the flows that target Xilinx, the commercial HLS tool delivers considerably more compact implementations than the academic tools (much smaller LUT consumption) since we anticipate it implements more technology-oriented optimizations. For all flows (including LEGUP), we observe that, in the performance-optimized flow, more resources are used to improve effectively performance.

V. DISCUSSON FROM THE TOOL PERSPECTIVE

In this section, we describe the results for the academic HLS tools from a tool-specific viewpoint and highlight techniques used to improve performance in each tool.

A. Bambu

BAMBU leverages GCC to perform classical code optimizations, such as loop unrolling, constant propagation, etc. To simplify the use of the tool for software designers, its interface has been designed such that the designer can use the same compilation flags and directives that would be given to GCC. In the *standard-optimization* case, the compiler optimization level passed to GCC is `-O3`, without any modifications to the source code of the benchmarks. In the *performance-optimized* study, the source code was modified only in case of *sobel*, where we used the same version modified by the LEGUP team. Loop unrolling was used for *adpcm*, *matrix* and *sha*. On three benchmarks (*gsm*, *matrix* and *sobel*), GCC vectorization produced a better wall-time, while function inlining was useful for *gsm*, *dfadd*, *dfsin*, *aes encrypt* and *decrypt*.

BAMBU's front-end phase implements also operation transformations that are specific for HLS, e.g. by transforming

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2015.2513673, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE IV: Standard-Optimization Performance Results. Fmax is Reported in MHz, Wall-clock in μs.

	Commercial			BAMBU			DWARV			LEGUP		
Benchmark	**Cycles**	**Fmax**	**Wall-clock**	**Cycles**	**Fmax**	**Wall-clock**	**Cycles**	**Fmax**	**Wall-clock**	**Cycles**	**Fmax**	**Wall-clock**
adpcm_encode	27250	281	96.87	11179	232	48.14	24454	183	133.67	7883	245	32.12
aes_encrypt	3976	345	11.54	1574	252	6.25	5135	201	25.60	1564	395	3.96
aes_decrypt	5461	322	16.95	2766	260	10.64	2579	255	10.11	7367	313	23.56
gsm	5244	347	15.12	2805	200	14.01	6866	186	36.90	3966	273	14.52
sha	197867	327	605.08	111762	259	431.62	71163	253	281.52	168886	250	676.90
blowfish	101010	397	254.65	57590	288	200.24	70200	251	280.03	75010	468	160.22
dfadd	552	332	1.66	404	275	1.47	465	215	2.16	650	252	2.58
dfdiv	2068	281	7.35	1925	222	8.65	2274	179	12.69	2046	183	11.20
dfmul	200	281	0.71	174	259	0.67	293	154	1.90	209	186	1.12
dfsin	57564	247	233.08	56021	223	251.09	64428	134	481.02	57858	189	305.79
jpeg	994945	208	4776.73	662380	217	3057.55	748707	ERR	ERR	1128109	220	5126.14
mips	4199	281	14.93	4043	259	15.60	8320	370	22.51	5989	487	12.30
motion	ERR	ERR	ERR	127	287	0.44	152	163	0.93	66	338	0.20
satd	87	383	0.23	27	232	0.12	57	134	0.42	46	288	0.16
sobel	45261481	330	137142.29	5983199	276	21665.16	23934323	340	70295.11	7561317	336	22502.58
bellmanford	2838	447	6.35	3218	227	14.17	2319	360	6.44	2444	332	7.37
matrix	363585	281	1292.54	198690	282	704.75	297026	391	759.79	101442	401	253.00
GEOMEAN	11918.22	321.6	37.06	6754.93	248.51	27.1821	10373.59	226.34	45.83	8039.75	292.514	27.48
GEOMEAN (ALL)				5681.2	246.67	23.03				6922.61	284.285	24.35

TABLE V: Performance-Optimized Results. Fmax is Reported in MHz, Wall-clock in μs.

	Commercial			BAMBU			DWARV			LEGUP		
Benchmark	**Cycles**	**Fmax**	**Wall-clock**	**Cycles**	**Fmax**	**Wall-clock**	**Cycles**	**Fmax**	**Wall-clock**	**Cycles**	**Fmax**	**Wall-clock**
adpcm_encode	12350	281	43.90	7077	258	27.40	9122	148	61.47	6635	348	19.06
aes_encrypt	3735	331	11.29	1485	249	5.96	3282	250	13.13	1191	408	2.92
aes_decrypt	3923	307	12.77	2585	254	10.17	2579	255	10.11	4847	319	15.19
gsm	3584	347	10.34	2128	180	11.83	7308	333	21.92	1931	262	7.36
sha	124339	329	377.87	51399	203	253.35	71163	253	281.52	81786	219	256.74
blowfish	96460	350	275.68	57590	288	200.24	70200	251	280.03	64480	536	120.32
dfadd	552	332	1.66	370	243	1.52	465	215	2.16	319	258	1.24
dfdiv	2068	281	7.35	1374	240	5.73	2846	263	10.83	942	161	5.85
dfmul	200	281	0.71	162	253	0.64	293	154	1.90	105	183	0.57
dfsin	57564	247	233.08	38802	233	166.69	90662	333	271.99	22233	135	165.02
jpeg	602725	209	2882.83	662380	217	3057.55	706151	ERR	ERR	1182092	255	4639.66
mips	4199	281	14.93	5783	411	14.06	8320	370	22.51	5989	487	12.30
motion	ERR	ERR	ERR	127	285	0.45	122	167	0.73	66	338	0.20
satd	27	497	0.05	36	442	0.08	54	473	0.11	42	289	0.15
sobel	2475541	330	7495.94	3641402	480	7585.04	3648547	287	12696.94	1565741	489	3201.92
bellmanford	2607	408	6.38	4779	509	9.38	2319	360	6.44	1036	418	2.48
matrix	16408	281	58.33	6178	238	25.90	36162	386	93.73	19003	345	55.01
GEOMEAN	6396.3	320.8	19.9	4704.6	283.9	16.6	7509.6	275.8	27.2	4185.8	299.9	13.6
GEOMEAN (ALL)				5089.0	279.5	18.2				4570.2	299.1	14.9

TABLE VI: Standard-Optimization Area Results.

	Commercial			BAMBU			DWARV			LEGUP		
Benchmark	***LUTp***	***BRAMB18***	***DSP48s***	***LUTp***	***BRAMB18***	***DSP48s***	***LUTp***	***BRAMB18***	***DSP48s***	***ALMs***	**M20K**	***DSPs***
adpcm_encode	4319	0	68	19931	52	64	5626	18	6	2490	0	43
aes_encrypt	5802	6	1	8485	4	0	15699	16	3	4263	8	0
aes_decrypt	6098	4	1	8747	4	1	12733	16	3	4297	14	0
gsm	5271	8	49	11864	10	75	6442	0	8	4311	1	51
sha	2161	16	0	4213	12	0	10012	0	0	6398	26	0
blowfish	2226	0	0	6837	0	0	7739	0	0	1679	0	0
dfadd	7409	0	0	7250	0	0	7334	0	0	2812	1	0
dfdiv	15107	0	24	11757	0	24	13934	1	40	4679	4	42
dfmul	3070	0	16	3430	0	16	14157	1	40	1464	1	28
dfsin	22719	0	43	21892	0	59	30616	43	43	9099	3	72
jpeg	16192	25	1	46757	154	26	ERR	ERR	ERR	16276	41	85
mips	1963	3	8	2501	0	8	3904	3	20	1319	0	15
motion	ERR	ERR	ERR	2776	2	0	45826	6	0	6788	0	0
satd	790	0	0	4425	0	0	1411	0	0	2004	0	0
sobel	792	0	6	3106	0	28	1160	0	12	1241	0	36
bellmanford	485	0	0	1046	0	0	633	0	0	493	0	0
matrix	175	0	3	551	0	3	471	0	3	225	0	2
GEOMEAN	2711.75	1.84	4.57	5253.60	2.15	5.30	5148.72	2.43	4.88	2197.66	2.01	5.67
GEOMEAN (ALL)				5754.49	2.76	5.28				2641.94	2.30	6.00

TABLE VII: Performance-Optimized Area Results.

	Commercial			BAMBU			DWARV			LEGUP		
Benchmark	***LUTp***	***BRAMB18***	***DSP48s***	***LUTp***	***BRAMB18***	***DSP48s***	***LUTp***	***BRAMB18***	***DSP48s***	***ALMs***	**M20K**	***DSPs***
adpcm_encode	5325	0	116	10546	2	81	13416	0	6	2903	0	57
aes_encrypt	5798	6	1	9793	2	1	15699	16	3	3199	0	0
aes_decrypt	6370	4	1	12927	2	3	12733	16	3	4894	18	0
gsm	8970	11	49	29646	16	316	6442	0	8	3442	3	59
sha	13105	16	0	14819	12	0	10012	0	0	28289	12	0
blowfish	3433	0	0	6799	0	0	7739	0	0	1648	0	0
dfadd	7409	0	0	6413	0	0	7334	0	0	3506	0	0
dfdiv	15107	0	24	7673	1	76	16209	1	40	16895	9	126
dfmul	3070	0	16	3001	0	16	14157	1	40	1866	0	28
dfsin	22719	0	43	21538	1	111	30616	43	43	10857	3	72
jpeg	16099	25	1	46757	154	26	ERR	ERR	ERR	16669	41	85
mips	1963	3	8	2305	0	8	3904	3	20	1319	0	15
motion	ERR	ERR	ERR	2678	1	0	49414	6	0	6788	0	0
satd	1704	0	0	2447	2	0	3037	0	0	1959	0	0
sobel	1015	0	3	722	0	0	2877	0	3	698	0	0
bellmanford	1127	0	0	717	0	0	633	0	0	528	1	0
matrix	3406	0	96	10531	0	384	7110	0	3	3747	0	68
GEOMEAN	4575.89	1.88	5.70	5925.67	1.71	7.95	7384.52	2.00	4.45	3159.64	1.92	6.25
GEOMEAN (ALL)				6385.85	2.16	7.54				3644.68	2.21	6.54

multiplications and divisions which are usually very expensive in hardware. BAMBU maps 64-bit divisions onto a C library function implementing the Newton-Raphson algorithm for the integer division. This leads to a higher number of DSPs required by *dfdiv* and *dfsin* in the *standard-optimzation* case. BAMBU also supports floating-point operations since it interfaces with FloPoCo library [56].

All functional units are pre-characterized for multiple combinations of target devices, bit-widths and pipeline stages. Hence, BAMBU implements a technology-aware scheduler to perform aggressive operation chaining and code motion. This reduces the total number of clock cycles, while respecting the given timing constraint. Trimming of the address bus was useful for *bellmanford*, *matrix*, *satd*, and *sobel*.

Finally, BAMBU adopts a novel architecture for memory accesses [11]. Specifically, BAMBU builds a hierarchical datapath directly connected to a dual-port BRAM whenever a local aggregated or a global scalar/aggregate data type is used by the kernel and whenever the accesses can be determined at compile time. In this case, multiple memory accesses can be performed in parallel. Otherwise, the memories are interconnected so that it is also possible to support dynamic resolution of the addresses. Indeed, the same memory infrastructure can be natively connected to external components (e.g. a local scratch-pad memory or cache) or directly to the bus to access off-chip memory. Finally, if the kernel has pointers as parameters, it assumes that the objects referred are allocated on dual-port BRAMs.

The optimized results obtained for *blowfish* and *jpeg* are the same obtained in the first study since we were not able to identify different options to improve the results.

B. DWARV

Since DWARV is based on CoSy [10], one of the main advantages is its flexibility to easily exploit standard and custom optimizations. The framework contains 255 transformations and optimizations passes available in the form of stand-alone *engines*. For the *standard-evaluation* experiment, the most important optimizations that DWARV uses are if-conversion, operation chaining, multiple memories and a simple (i.e. analysis based only on standard integer types) bit-width analysis. For the *performance-optimized* runs, pragmas were added to enable loop unrolling. However, not all framework optimizations are yet fully integrated in the HLS flow. One of the DWARV restrictions is that it does not support global variables. As a result, the CHStone benchmarks, which rely heavily on global variables, had to be rewritten to transform global variables to function parameters passed by reference. Besides the effort needed to rewrite code accessing global memory, some global optimizations across functions are not considered. Another limitation is a mismatch between the clock period targeted by operation-chaining and the selection of IP cores in the target technology (e.g. for a divider unit), which are not (re)generated on request based on a target frequency. Operation chaining is set to a specific target frequency for each benchmark (as shown in Table II). However this can differ significantly from that achievable within the instantiated IP cores available in DWARV's IP library, as shown for example in the *dfxxx* kernels. DWARV targets mostly small and medium size kernels. It thus generates a central FSM and always maps local arrays to distributed logic. This is a problem for large kernels such as the *jpeg* benchmark, which could not be mapped in the available area on the target platform. Another minor limitation is the transformation – in the compiler backend – of switch constructs to if-else constructs. Generating lower-level switch constructs would improve the *aes, mips, jpeg* kernels, that contain multiple switch statements.

C. LegUp

Several methods exist for optimizing LEGUP-produced circuits: automatic LLVM compiler optimizations [12], user-defined directives for activating various hardware specific features, and source code modifications. Since LEGUP is built within LLVM, users can utilize LLVM optimization passes

with minimal effort. In the context of hardware circuits, for the *performance-optimized* runs, function inlining and loop unrolling provided benefits across multiple benchmarks. Function inlining allows the hardware scheduler to exploit more instruction-level parallelism and simplify the FSM. Similarly, loop unrolling exposes more parallelism across loop iterations. The performance boost associated with inlining and unrolling generally comes at the cost of increased area.

LEGUP also offers many hardware optimizations that users can activate by means of `tcl` directives, such as activating loop pipelining or changing the target clock period. Loop pipelining allows consecutive iterations of a loop to begin execution before the previous iteration has completed, reducing the overall number of clock cycles. Longer clock periods permit more chaining, reducing cycle latency. If the reduction in cycle latency does not exceed the amount by which the clock period lengthens, wall-clock time will be also improved.

Manual source code modifications can be made to assist LEGUP in inferring parallelism within the program. One such modification is to convert single-threaded execution to multi-threaded execution using pthreads/OpenMP, whereby LEGUP synthesizes the multiple parallel threads into parallel hardware accelerators. This optimization was applied for all of the *df* benchmarks. In the *df* benchmarks, a set of inputs is applied to a kernel in a data-parallel fashion – there are no dependencies between the inputs. Such a situation is particularly desirable for LEGUP's multi-threading synthesis: multiple identical hardware kernels are instantiated, each operating in parallel on disjoint subsets of the input data.

VI. CONCLUDING REMARKS

To the authors' knowledge, this paper represents the first broad evaluation of several HLS tools. We presented an extensive survey and categorization for past and present hardware compilers. We then described the optimizations on which recent and ongoing research in the HLS community is focussed. We experimentally evaluated three academic HLS tools, BAMBU, DWARV and LEGUP, against a commercial tool. The methodology aims at providing a fair comparison of tools, even if they are built within different compiler frameworks and target different FPGA families. The results shows that each HLS tool can significantly improve the performance with benchmark-specific optimizations and constraints. However, software engineers need to take into account that optimizations that are necessary to realize high performance in hardware (e.g. enabling loop pipelining, removing control flow, etc.) differ significantly from software-oriented ones (e.g. data re-organization for cache locality).

Overall, the performance results showed that academic and commercial HLS tools are not drastically far apart in terms of quality, and that no single tool producted the best results for all benchmarks. Obviously, despite this, it should nevertheless be noted that the commercial compiler supports more features, allowing multiple input and output languages, the customization of the generated kernels in terms of interface types, memory bank usage, throughput, etc., while at the same time also being more robust than the academic tools.

REFERENCES

[1] S. Borkar and A. A. Chien. *The Future of Microprocessors*. Communications of the ACM, 54:67–77, May 2011.

[2] P. Coussy and A. Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008.

[3] H.-Y. Liu, M. Petracca, and L. P. Carloni, *Compositional System-Level Design Exploration with Planning of High-level Synthesis*. In IEEE/ACM DATE, pp. 641–646, 2012.

[4] A. Putnam, A. Caulfield, et al. *A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services.* IEEE/ACM ISCA, pages 13-24, 2014.

[5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. *Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks*. In ACM FPGA, pages 161-170. 2015.

[6] W. Meeus, K. Van Beeck, T. Goedem, J. Meel, D. Stroobandt. *An Overview of Todays High-Level Synthesis Tools*. In Design Automation for Embedded Systems, 2012, pp. 31-51.

[7] R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, K. Bertels. *DWARV 2.0: A CoSy-based C-to-VHDL Hardware Compiler*. In FPL, pp. 619-622. 2012.

[8] C. Pilato and F. Ferrandi. *Bambu: A Modular Framework for the High Level Synthesis of Memory-intensive Applications*. In FPL, pp. 1–4, 2013.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, T. Czajkowski. *LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems*. In ACM FPGA, pp. 33-36. 2011.

[10] ACE- Associated Compiler Experts. *CoSy*. [Online]. Available: http://www.ace.nl

[11] C. Pilato, F. Ferrandi, and D. Sciuto. *A Design Methodology to Implement Memory Accesses in High-Level Synthesis*. In IEEE/ACM CODES+ISSS, pp. 49–58, 2011.

[12] C. Lattner, V. Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In IEEE/ACM CGO, pp75-88, 2004.

[13] K. Wakabayashi, T. Okamoto. *C-based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective*. In IEEE TCAD, vol. 19, no. 12, pp. 1507-1522, 2000.

[14] BlueSpec Inc. *High-Level Synthesis Tools*. [Online]. Available: http://bluespec.com/high-level-synthesis-tools.html

[15] R. Nikhil. *Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications*. In IEEE/ACM MEMOCODE, pp. 69-70. 2004.

[16] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R.R. Taylor, R. Reed. *PipeRench: A Reconfigurable Architecture and Compiler*. In IEEE Computer, pp. 70-77, 2000.

[17] N. Kavvadias, K. Masselos. *Automated Synthesis of FSMD-Based Accelerators for Hardware Compilation*. In IEEE ASAP, pp. 157-160. 2012.

[18] Impulse Accelerated Technologies. *Impulse CoDeveloper C-to-FPGA Tools*. [Online]. Available: http://www.impulseaccelerated.com/products_universal.htm

[19] Mentor Graphics. *DK Design Suite: Handel-C to FPGA for Algorithm Design*. [Online]. Available: http://www.mentor.com/products/fpga/handel-c/dk-design-suite

[20] W.A. Najjar, W. Bohm, B.A. Draper, J. Hammes, R. Rinker, J.R. Beveridge, M. Chawathe, C. Ross. *High-Level Language Abstraction for Reconfigurable Computing*. In IEEE Computer 2003. pp. 63-69.

[21] T. Callahan, J. Hauser, R. John, J. Wawrzynek. *The Garp Architecture and C Compiler*. In IEEE Computer, pp. 62-69, 2000.

[22] M.B. Gokhale, J.M. Stone. *NAPA C: Compiling for a Hybrid RISC/FPGA Architecture*. In IEEE FCCM, pp. 126-135, 1998.

[23] Y Explorations. *eXCite: C to RTL Behavioral Synthesis*. [Online]. Available: http://www.yxi.com/products.php

[24] J. Villarreal, A. Park, W. Najjar, R. Halstead. *Designing Modular Hardware Accelerators in C with ROCCC 2.0*. In IEEE FCCM, pp. 127-134. 2010.

[25] Calypto Design Systems. *Catapult: Product Family Overview*. [Online]. Available: http://calypto.com/en/products/catapult/overview

[26] Cadence. *C-to-Silicon Compiler*. [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx

[27] S. Gupta, N. Dutt, R. Gupta, A. Nicolau. *SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations*. In VLSI Design, pp. 461-466, 2003.

[28] Altium. *Altium Designer: A Unified Solution*. [Online]. Available: http://www.altium.com/en/products/altium-designer

[29] Universite de Bretagne-Sud. *GAUT – High-Level Synthesis Tool from C to RTL*. [Online]. Available: http://hls-labsticc.univ-ubs.fr/

[30] J.L. Tripp, M.B. Gokhale, K.D. Peterson. *Trident: From High-Level Language to Hardware Circuitry*. In IEEE Computer, pp. 28-37, 2007.