# High-Level Synthesis for FPGAs: From Prototyping to Deployment

Jason Cong[1,2], *Fellow, IEEE,* Bin Liu[1,2], Stephen Neuendorffer[3], *Member, IEEE,* Juanjo Noguera[3], Kees Vissers[3], *Member, IEEE* and Zhiru Zhang[1], *Member, IEEE*

[1]AutoESL Design Technologies, Inc.
[2]University of California, Los Angeles
[3]Xilinx, Inc.

***Abstract*—Escalating System-on-Chip design complexity is pushing the design community to raise the level of abstraction beyond RTL. Despite the unsuccessful adoptions of early generations of commercial high-level synthesis (HLS) systems, we believe that the tipping point for transitioning to HLS methodology is happening now, especially for FPGA designs. The latest generation of HLS tools has made significant progress in providing wide language coverage and robust compilation technology, platform-based modeling, advancement in core HLS algorithms, and a domain-specific approach. In this paper we use AutoESL's AutoPilot HLS tool coupled with domain-specific system-level implementation platforms developed by Xilinx as an example to demonstrate the effectiveness of state-of-art C-to-FPGA synthesis solutions targeting multiple application domains. Complex industrial designs targeting Xilinx FPGAs are also presented as case studies, including comparison of HLS solutions versus optimized manual designs.**

***Index Terms*—Domain-specific design, field-programmable gate array (FPGA), high-level synthesis (HLS), quality of results (QoR).**

## I. Introduction

THE RAPID INCREASE of complexity in System-on-a-Chip (SoC) design has encouraged the design community to seek design abstractions with better productivity than RTL. Electronic system-level (ESL) design automation has been widely identified as the next productivity boost for the semiconductor industry, where HLS plays a central role, enabling the automatic synthesis of high-level, untimed or partially timed specifications (such as in C or SystemC) to a low-level cycle-accurate register-transfer level (RTL) specifications for efficient implementation in ASICs or FPGAs. This synthesis can be optimized taking into account the performance, power, and cost requirements of a particular system.

Despite the past failure of the early generations of commercial HLS systems (started in the 1990s), we see a rapidly growing demand for innovative, high-quality HLS solutions for the following reasons:

- **Embedded processors are in almost every SoC**: With the coexistence of micro-processors, DSPs, memories and custom logic on a single chip, more software elements are involved in the process of designing a modern embedded system. An automated HLS flow allows designers to specify design functionality in high-level programming languages such as C/C++ for both embedded software and customized hardware logic on the SoC. This way, they can quickly experiment with different hardware/software boundaries and explore various area/power/performance tradeoffs from a single common functional specification.
- **Huge silicon capacity requires a higher level of abstraction**: Design abstraction is one of the most effective methods for controlling complexity and improving design productivity. For example, the study from NEC [90] shows that a 1M-gate design typically requires about 300K lines of RTL code, which cannot be easily handled by a human designer. However, the code density can be easily reduced by 7X to 10X when moved to high-level specification in C, C++, or SystemC. In this case, the same 1M-gate design can be described in 30K to 40K lines of lines of behavioral description, resulting in a much reduced design complexity.
- **Behavioral IP reuse improves design productivity:** In addition to the line-count reduction in design specifications, behavioral synthesis has the added value of allowing efficient reuse of behavioral IPs. As opposed to RTL IP which has fixed microarchitecture and interface protocols, behavioral IP can be retargeted to different implementation technologies or system requirements.
- **Verification drives the acceptance of high-level specification**: Transaction-level modeling (TLM) with SystemC [107] or similar C/C++ based extensions has become a very popular approach to system-level verification [35]. Designers commonly use SystemC TLMs to describe virtual software/hardware platforms, which serve three important purposes: early embedded software development, architectural modeling and exploration, and functional verification. The wide availability of SystemC functional models directly drives the need for SystemC-based HLS solutions, which can automatically generate RTL code through a series of formal constructive transformations. This avoids slow and error-prone manual RTL re-coding, which is the standard practice in the industry today.
- **Trend towards extensive use of accelerators and heterogeneous SoCs**: Many SoCs, or even CMPs (chip multi-processors) move towards inclusion of many accelerators (or algorithmic blocks), which are built with custom architectures, largely to reduce power compared to using multiple programmable processors. According to ITRS prediction [109], the number of on-chip accelerators will reach 3000 by 2024. In FPGAs, custom

architecture for algorithmic blocks provides higher performance in a given amount of FPGA resources than synthesized soft processors. These algorithmic blocks are particularly appropriate for HLS.

Although these reasons for adopting HLS design methodology are common to both ASIC and FPGA designers, we also see additional forces that push the FPGA designers for faster adoption of HLS tools.

- **Less pressure for formal verification:** The ASIC manufacturing cost in nanometer IC technologies is well over $1M [109]. There is tremendous pressure for the ASIC designers to achieve first tape-out success. Yet formal verification tools for HLS are not mature, and simulation coverage can be limited for multi-million gate SOC designs. This is a significant barrier for HLS adoption in the ASIC world. However, for FPGA designs, in-system simulation is possible with much wider simulation coverage. Design iterations can be done quickly and inexpensively without huge manufacturing costs.
- **Ideal for platform-based synthesis:** Modern FPGAs embed many pre-defined/fabricated IP components, such as arithmetic function units, embedded memories, embedded processors, and embedded system buses. These pre-defined building blocks can be modeled precisely ahead of time for each FPGA platform and, to a large extent, confine the design space. As a result, it is possible for modern HLS tools to apply a platform-based design methodology [51] and achieve higher quality of results (QoR).
- **More pressure for time-to-market:** FPGA platforms are often selected for systems where time-to-market is critical, in order to avoid long chip design and manufacturing cycles. Hence, designers may accept increased performance, power, or cost in order to reduce design time. As shown in Section IX, modern HLS tools put this tradeoff in the hands of a designer allowing significant reduction in design time or, with additional effort, quality of result comparable to hand-written RTL.
- **Accelerated or reconfigurable computing calls for C/C++ based compilation/synthesis to FPGAs**: Recent advances in FPGAs have made reconfigurable computing platforms feasible to accelerate many high-performance computing (HPC) applications, such as image and video processing, financial analytics, bioinformatics, and scientific computing applications. Since RTL programming in VHDL or Verilog is unacceptable to most application software developers, it is essential to provide a highly automated compilation/synthesis flow from C/C++ to FPGAs.

As a result, a growing number of FPGA designs are produced using HLS tools. Some example application domains include 3G/4G wireless systems [38][81], aerospace applications [75], image processing [27], lithography simulation [13], and cosmology data analysis [52]. Xilinx is also in the process of incorporating HLS solutions in their Video Development Kit [116] and DSP Develop Kit [97] for all Xilinx customers.

This paper discusses the reasons behind the recent success in deploying HLS solutions to the FPGA community. In Section II we review the evolution of HLS systems and summarize the key lessons learned. In Sections IV-VIII, using a state-of-art HLS tool as an example, we discuss some key reasons for the wider adoption of HLS solutions in the FPGA design community, including wide language coverage and robust compilation technology, platform-based modeling, advancement in core HLS algorithms, improvements on simulation and verification flow, and the availability of domain-specific design templates. Then, in Section IX, we present the HLS results on several real-life industrial designs and compare with manual RTL implementations. Finally, in Section X, we conclude the paper with discussions of future challenges and opportunities.

## II. Evolution of High-Level Synthesis for FPGA

In this section we briefly review the evolution of high-level synthesis by looking at representative tools. Compilers for high-level languages have been successful in practice since the 1950s. The idea of automatically generating circuit implementations from high-level behavioral specifications arises naturally with the increasing design complexity of integrated circuits. Early efforts (in the 1980s and early 1990s) on high-level synthesis were mostly research projects, where multiple prototype tools were developed to call attention to the methodology and to experiment with various algorithms. Most of those tools, however, made rather simplistic assumptions about the target platform and were not widely used. Early commercialization efforts in the 1990s and early 2000s attracted considerable interest among designers, but also failed to gain wide adoption, due in part to usability issues and poor quality of results. More recent efforts in high-level synthesis have improved usability by increasing input language coverage and platform integration, as well as improving quality of results.

### *A. Early Efforts*

Since the history of HLS is considerably longer than that of FPGAs, most early HLS tools targeted ASIC designs. A pioneering high-level synthesis tool, CMU-DA, was built by researchers at Carnegie Mellon University in the 1970s [29][71]. In this tool the design is specified at behavior level using the ISPS (Instruction Set Processor Specification) language [4]. It is then translated into an intermediate data-flow representation called the Value Trace [79] before producing RTL. Many common code-transformation techniques in software compilers, including dead-code elimination, constant propagation, redundant sub-expression elimination, code motion, and common sub-expression extraction could be performed. The synthesis engine also included many steps familiar in hardware synthesis, such as datapath allocation, module selection, and controller generation. CMU-DA also supported hierarchical design and included a simulator of the original ISPS language. Although many of the methods used were very preliminary, the

innovative flow and the design of toolsets in CMU-DA quickly generated considerable research interest.

In the subsequent years in the 1980s and early 1990s, a number of similar high-level synthesis tools were built, mostly for research. Examples of academic efforts include the ADAM system developed at the University of Southern California [37][46], HAL developed at Bell-Northern Research [72], MIMOLA developed at University of Kiel, Germany [62], the Hercules/Hebe high-level synthesis system (part of the Olympus system) developed at Stanford University [24][25] [55], the Hyper/Hyper-LP system developed at University of California, Berkeley [10][77]. Industry efforts include Cathedral/Cathedral-II and their successors developed at IMEC [26], the IBM Yorktown Silicon Compiler [11] and the GM BSSC system [92], among many others. Like CMU-DA, these tools typically decompose the synthesis task into a few steps, including code transformation, module selection, operation scheduling, datapath allocation, and controller generation. Many fundamental algorithms addressing these individual problems were also developed. For example, the list scheduling algorithm and its variants are widely used to solve scheduling problems with resource constraints [70]; the force-directed scheduling algorithm developed in HAL [73] is able to optimize resource requirements under a performance constraint; the path-based scheduling algorithm in the Yorktown Silicon Compiler is useful to optimize performance with conditional branches [12]. The Sehwa tool in ADAM is able to generate pipelined implementations and explore the design space by generating multiple solutions [69]. The relative scheduling technique developed in Hebe is an elegant way to handle operations with unbounded delay [56]. Conflict-graph coloring techniques were developed and used in several systems to share resources in the datapath [57][72].

These early high-level tools often used custom languages for design specification. Besides the ISPS language used in CMD-DA, a few other languages were notable. HardwareC is a language designed for use in the Hercules system [54]. Based on the popular C programming language, it supports both procedural and declarative semantics and has built-in mechanisms to support design constraints and interface specifications. This is one of the earliest C-based hardware synthesis languages for high-level synthesis and is interesting to compare with similar languages later. The Silage language used in Cathedral/Cathedral-II was specifically designed for the synthesis of digital signal processing hardware [26]. It has built-in support for customized data types, and allows easy transformations [77][10]. The Silage language, along with the Cathedral-II tool, represented an early domain-specific approach in high-level synthesis.

These early research projects helped to create a basis for algorithmic synthesis with many innovations, and some were even used to produce real chips. However, these efforts did not lead to wide adoption among designers. A major reason is that the methodology of using RTL synthesis was not yet widely accepted at that time and RTL synthesis tools were not yet mature. Thus, high-level synthesis, built on top of RTL synthesis, did not have a sound foundation in practice. In addition, simplistic assumptions were often made in these early systems—many of them were "technology independent" (such as Olympus), and inevitably led to suboptimal results.

With improvements in RTL synthesis tools and the wide adoption of RTL-based design flows in the 1990s, industrial deployment of high-level synthesis tools became more practical. Proprietary tools were built in major semiconductor design houses including IBM [5], Motorola [58], Philips [61], and Simens [6]. Major EDA vendors also began to provide commercial high-level synthesis tools. In 1995, Synopsys announced Behavioral Compiler [88], which generates RTL implementations from behavioral HDL code and connects to downstream tools. Similar tools include Monet from Mentor Graphics [33] and Visual Architect from Cadence [43]. These tools received wide attention, but failed to widely replace RTL design. One reason is due to the use of behavioral HDLs as the input language, which is not popular among algorithm and system designers.

### *B. Recent efforts*

Since 2000, a new generation of high-level synthesis tools has been developed in both academia and industry. Unlike many predecessors, most of these tools focus on using C/C++ or C-like languages to capture design intent. This makes the tools much more accessible to algorithm and system designers compared to previous tools that only accept HDL languages. It also enables hardware and software to be built using a common model, facilitating software/hardware co-design and co-verification. The use of C-based languages also makes it easy to leverage the newest technologies in software compilers for parallelization and optimization in the synthesis tools.

In fact, there has been an ongoing debate on whether C-based languages are proper choices for HLS [31][78]. Despite the many advantages of using C-based languages, opponents often criticize C/C++ as languages only suitable for describing sequential software that runs on microprocessors. Specifically, the deficiencies of C/C++ include the following:

(i) Standard C/C++ lack built-in constructs to explicitly specify bit accuracy, timing, concurrency, synchronization, hierarchy, etc., which are critical to hardware design.

(ii) C and C++ have complex language constructs, such as pointers, dynamic memory management, recursion, polymorphism, etc., which do have efficient hardware counterparts and lead to difficulty in synthesis.

To address these deficiencies, modern C-based HLS tools have introduced additional language extensions and restrictions to make C inputs more amenable to hardware synthesis. Common approaches include both restriction to a synthesizable subset that discourages or disallows the use of dynamic constructs (as required by most tools) and introduction of hardware-oriented language extensions (HardwareC [54], SpecC [34], Handel-C [95]), libraries (SystemC [107]), and compiler directives to specify concurrency, timing, and other constraints. For example, Handel-C allows the user to specify clock boundaries explicitly in the source code. Clock edges and events can also be explicitly specified in SpecC and SystemC. Pragmas and

directives along with a subset of ANSI C/C++ are used in many commercial tools. An advantage of this approach is that the input program can be compiled using standard C/C++ compilers without change, so that such a program or a module of it can be easily moved between software and hardware and co-simulation of hardware and software can be performed without code rewriting. At present, most commercial HLS tools use some form of C-based design entry, although tools using other input languages (e.g., BlueSpec [102], Esterel [30], Matlab [42], etc.) also exist.

Another notable difference between the new generation of high-level synthesis tools and their predecessors is that many tools are built targeting implementation on FPGA. FPGAs have continually improved in capacity and speed in recent years, and their programmability makes them an attractive platform for many applications in signal processing, communication, and high-performance computing. There has been a strong desire to make FPGA programming easier, and many high-level synthesis tools are designed to specifically target FPGAs, including ASC [64], CASH [9], C2H from Altera [98], DIME-C from Nallatech [112], GAUT [22], Handel-C compiler (now part of Mentor Graphics DK Design Suite) [95], Impulse C [74], ROCCC [87][39], SPARK [41][40], Streams-C compiler [36], and Trident [82][83], . ASIC tools also commonly provide support for targeting an FPGA tool flow in order to enable system emulation.

Among these high-level synthesis tools, many are designed to focus on a specific application domain. For example, the Trident compiler, developed at Los Alamos National Lab, is an open-source tool focusing on the implementation of floating-point scientific computing applications on FPGA. Many tools, including GAUT, Streams-C, ROCCC, ASC, and Impulse C, target streaming DSP applications. Following the tradition of Cathedral, these tools implement architectures consisting of a number of modules connected using FIFO channels. Such architectures can be integrated either as a standalone DSP pipeline, or integrated to accelerate code running on a processor (as in ROCCC).

As of 2010, major commercial C-based high-level synthesis tools include AutoESL's AutoPilot [94] (originated from UCLA xPilot project [17]), Cadence's C-to-Silicon Compiler [3][103], Forte's Cynthesizer [65], Mentor's Catapult C [7], NEC's Cyber Workbench [89][91], and Synopsys Synphony C [115] (formerly Synfora's PICO Express, originated from a long range research effort in HP Labs [49]).

### C. *Lessons Learned*

Despite extensive development efforts, most commercial HLS efforts have failed. We believe that past failures are due to one or several of the following reasons:

- **Lack of comprehensive design language support**: The first generation of the HLS synthesis tools could not synthesize high-level programming languages. Instead, untimed or partially timed behavioral HDL was used. Such design entry marginally raised the abstraction level, while imposing a steep learning curve on both software and hardware developers.

  Although early C-based HLS technologies have considerably improved the ease of use and the level of design abstraction, many C-based tools still have glaring deficiencies. For instance, C and C++ lack the necessary constructs and semantics to represent hardware attributes such as design hierarchy, timing, synchronization, and explicit concurrency. SystemC, on the other hand, is ideal for system-level specification with software/hardware co-design. However, it is foreign to algorithmic designers and has slow simulation speed compared to pure ANSI C/C++ descriptions. Unfortunately, most early HLS solutions commit to only one of these input languages, restricting their usage to niche application domains.
- **Lack of reusable and portable design specification**: Many HLS tools have required users to embed detailed timing and interface information as well as the synthesis constraints into the source code. As a result, the functional specification became highly tool-dependent, target-dependent, and/or implementation-platform dependent. Therefore, it could not be easily ported to alternative implementation targets.
- **Narrow focus on datapath synthesis:** Many HLS tools focus primarily on datapath synthesis, while leaving other important aspects unattended, such as interfaces to other hardware/software modules and platform integration. Solving the system integration problem then becomes a critical design bottleneck, limiting the value in moving to a higher-level design abstraction for IP in a design.
- **Lack of satisfactory quality of results (QoR)**: When early generations of HLS tools were introduced in the mid-1990s to early 2000s, the EDA industry was still struggling with timing closure between logic and physical designs. There was no dependable RTL to GDSII foundation to support HLS, which made it difficult to consistently measure, track, and enhance HLS results. Highly automated RTL to GDSII solutions only became available in late 2000s (e.g., provided by the IC Compiler from Synopsys [114] or the BlastFusion/Talus from Magma [111]). Moreover, many HLS tools are weak in optimizing real-life design metrics. For example, the commonly used algorithms mainly focus on reducing functional unit count and latency, which do not necessarily correlate to actual silicon area, power, and performance. As a result, the final implementation often fails to meet timing/power requirements. Another major factor limiting quality of result was the limited capability of HLS tools to exploit performance-optimized and power-efficient IP blocks on a specific platform, such as the versatile DSP blocks and on-chip memories on modern FPGA platforms. Without the ability to match the QoR achievable with an RTL design flow, most designers were unwilling to explore potential gains in design productivity.
- **Lack of a compelling reason/event to adopt a new design methodology:** The first-generation HLS tools

were clearly ahead of their time, as the design complexity was still manageable at the register transfer level in late 1990s. Even as the second-generation of HLS tools showed interesting capabilities to raise the level of design abstraction, most designers were reluctant to take the risk of moving away from the familiar RTL design methodology to embrace a new unproven one, despite its potential large benefits. Like any major transition in the EDA industry, designers needed a compelling reason or event to push them over the "tipping point," i.e., to adopt the HLS design methodology.

Another important lesson learned is that tradeoffs must be made in the design of the tool. Although a designer might wish for a tool that takes any input program and generates the "best" hardware architecture, this goal is not generally practical for HLS to achieve. Whereas compilers for processors tend to focus on local optimizations with the sole goal of increasing performance, HLS tools must automatically balance performance and implementation cost using global optimizations. However, it is critical that these optimizations be carefully implemented using scalable and predictable algorithms, keeping tool runtimes acceptable for large programs and the results understandable by designers. Moreover, in the inevitable case that the automatic optimizations are insufficient, there must be a clear path for a designer to identify further optimization opportunities and execute them by rewriting the original source code.

Hence, it is important to focus on several design goals for a high-level synthesis tool:

1. Capture designs at a bit-accurate, algorithmic level in C code. The code should be readable by algorithm specialists.
2. Effectively generate efficient parallel architectures with minimal modification of the C code, for parallelizable algorithms.
3. Allow an optimization-oriented design process, where a designer can improve the performance of the resulting implementation by successive code modification and refactoring.
4. Generate implementations that are competitive with synthesizable RTL designs after automatic and manual optimization.

We believe that the tipping point for transitioning to HLS methodology is happening now, given the reasons discussed in Section I and the conclusions by others [14][84]. Moreover, we are pleased to see that the latest generation of HLS tools has made significant progress in providing wide language coverage and robust compilation technology, platform-based modeling, and advanced core HLS algorithms. We shall discuss these advancements in more detail in the next few sections.

## III. Case Study of State-of-Art of High-Level Synthesis for FPGAs

AutoPilot is one of the most recent HLS tools, and is representative of the capabilities of the state-of-art commercial HLS tools available today. Figure 1 shows the AutoESL AutoPilot development flow targeting Xilinx FPGAs. AutoPilot accepts synthesizable ANSI C, C++, and OSCI SystemC (based on the synthesizable subset of the IEEE-1666 standard [113]) as input and performs advanced platform-based code transformations and synthesis optimizations to generate optimized synthesizable RTL.

AutoPilot outputs RTL in Verilog, VHDL or cycle-accurate SystemC for simulation and verification. To enable automatic co-simulation, AutoPilot creates test bench wrappers and transactors in SystemC so that designers can leverage the original test framework in C/C++/SystemC to verify the correctness of the RTL output. These SystemC wrappers connect high-level interfacing objects in the behavioral test bench with pin-level signals in RTL. AutoPilot also generates appropriate simulation scripts for use with 3rd-party RTL simulators. Thus designers can easily use their existing simulation environment to verify the generated RTL.

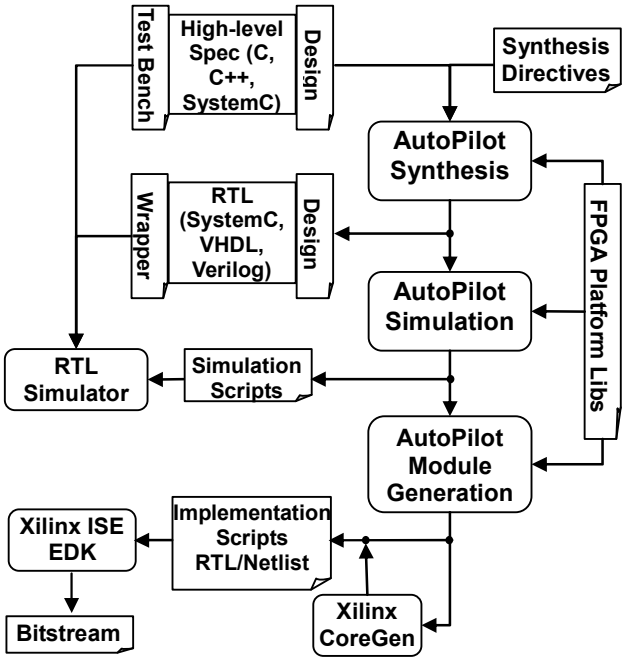


Figure 1. AutoESL and Xilinx C-to-FPGA design flow.

In addition to generating RTL, AutoPilot also creates synthesis reports that estimate FPGA resource utilization, as well as the timing, latency and throughput of the synthesized design. The reports include a breakdown of performance and area metrics by individual modules, functions and loops in the source code. This allows users to quickly identify specific areas for QoR improvement and then adjust synthesis directives or refine the source design accordingly.

Finally, the generated HDL files and design constraints feed into the Xilinx RTL tools for implementation. The Xilinx ISE tool chain (such as CoreGen, XST, PAR, etc.) and Embedded Development Kit (EDK) are used to transform that RTL implementation into a complete FPGA implementation in the form of a bitstream for programming the target FPGA platform.

## IV. SUPPORT OF HIGH-LEVEL PROGRAMMING MODELS

### A. Robust support of C/C++ based synthesis

Comprehensive language coverage is essential to enabling wide acceptance of C/C++ based design and synthesis. The reasons are twofold:

- **Reduced verification effort:** A broad synthesizable subset minimizes the required code changes to convert the reference C source into a synthesizable specification. This effectively improves the design productivity and reduces or eliminates the additional verification effort to ensure equivalence between the synthesizable code and the original design.
- **Improved design quality:** Comprehensive language support allows designers to take full advantage of rich C/C++ constructs to maximize simulation speed, design modularity and reusability, as well as synthesis QoR.

However, it is quite challenging to compile an input specification in software C language, which is known for its highly flexible syntax and semantic ambiguities, into a well-structured and well-optimized hardware described in HDL.

Table 1. Useful language features for effective C/C++-based design and synthesis.

| Language | Constructs | Benefits |
|---|---|---|
| C | Arbitrary-precision integer types | Bit-accurate design QoR |
| | Floating-point types | Floating-point arithmetic |
| | Function calls | Modular design hierarchy |
| | Pointers | Efficiency and flexibility |
| | Structs & unions | Data encapsulation |
| C++ | Fixed-point types | Fixed-point arithmetic Accuracy-cost tradeoff |
| | Templates | Parameterizable design |
| | Classes | Object-oriented modeling (encapsulation, inheritance, polymorphism, etc.) |
| SystemC | Modules & processes | Coarse-grained concurrency |
| | Clocks | Custom protocol Multi-clock design |
| | TLM | Fast simulation |

In fact, many early C-based synthesis tools only handle a very limited language subset, which typically includes the native integer data types (e.g., *char*, *short*, *int*, etc.), one-dimensional arrays, *if-then-else* conditionals, and *for* loops. Such language coverage is far from sufficient to allow complex large-scale designs. As shown in Table 1, supporting more advanced language features in C, C++ and SystemC is critical to raising the level of design abstraction and enabling efficient high-level synthesis.

AutoPilot accepts three standard C-based design entries in ANSI C, C++ and SystemC. It provides robust synthesis technologies to efficiently handle different aspects of the C/C++ language, such as data type synthesis (for both primitive and composite types), pointer synthesis, memory synthesis, control synthesis, loop synthesis, modular hierarchy synthesis (for function, class, and concurrent modules), and interface synthesis (for function parameters and global variables).

Designers can fully control the data precisions of a C/C++ specification. AutoPilot directly supports single- and double-precision floating-point types and efficiently utilizes the floating-point IPs provided by the FPGA platforms. Common floating-point math routines (e.g., square root, exponentiation, logarithm, etc.) can be mapped to high-quality platform-specific IPs.

In addition, AutoPilot has the capabilities to simulate and synthesize arbitrary-precision integers (*ap_int*) and fixed-point data types (*ap_fixed*). The arbitrary-precision fixed-point (*ap_fixed*) data types support all common algorithmic operations. With this library, designers can explore the accuracy and cost tradeoff by modifying the resolution and fixed-point location and experimenting with various quantization and saturation modes.

AutoPilot also supports the OCSI synthesizable subset [113] for SystemC synthesis. Designers can make use of SystemC bit-accurate data types (sc_int/sc_uint, sc_bigint/sc_biguint, and sc_fixed/sc_ufixed) to define the data precisions. Multi-module hierarchical designs can be specified and synthesized with multiple concurrent processes running inside each module.

### B. Use of state-of-the-art compiler technologies

AutoPilot tightly integrates the LLVM compiler infrastructure [59][110] to leverage leading-edge compiler technologies. LLVM features a GCC-based C/C++ front end called *llvm-gcc* and a newly developed source code front end for C/C++ and Object C/C++ called *Clang*, a virtual instruction set based on a type-safe static single-assignment (SSA) form [23], a rich set of code analyses and transformation passes, and various back ends for common target machines.

AutoPilot uses the *llvm-gcc* front end to obtain an intermediate representation (IR) based on the LLVM instruction set. On top of this IR, AutoPilot performs a variety of compiler transformations to aggressively optimize the input specification. The optimization focuses on reducing code complexity and redundancy, maximizing data locality, and exposing parallelism.

In particular, the following classes of transformations and analyses have shown to be very useful for hardware synthesis:

- SSA-based code optimizations such as constant propagation, dead code elimination, and redundant code elimination based on global value numbering [2].
- Expression rewriting such as strength reduction and arithmetic simplification to replace expensive operations and expressions with simpler ones (e.g., $x\%2^n = x\&(2^n-1)$, $3*x-x = x<<1$).
- Range analysis and bitwidth analysis [80][21] that

extract and propagate the value range information throughout the program to reduce bitwidths of variables and operations.
- Sophisticated alias analysis and memory dependence analysis [50] that analyzes data and control dependences to discover parallelism between pointer and array accesses.
- Memory optimizations such as memory reuse, array scalarization, and array partitioning [19] to reduce the number of memory accesses and improve memory bandwidth.
- Loop transformations such as unrolling, loop fusion, and loop rotation to expose loop-level parallelism [50].
- Function optimizations such as inlining and pointer-to-scalar argument promotion to enable code optimization across the function boundaries.

It is worth noting that the LLVM-based IR is in a language-agnostic format. In other words, the code can be optimized without considering the source language. As a result, the same set of analyses and optimizations on this representation can be shared and taken advantage of by many different language front ends.

Furthermore, unlike other conventional C/C++ compilers, which are typically designed to optimize with the native data types (e.g., *char*, *short*, *int*, *long*, etc.), LLVM and AutoPilot compilation and transformation procedures are fully bit accurate. This is a significant advantage for hardware synthesis since bit-level redundancy and parallelism can be well optimized and well exploited [93].

## V. Platform-based approach

### A. Platform modeling for Xilinx FPGAs

AutoPilot uses detailed target platform information to carry out informed and target-specific synthesis and optimization. The platform specification describes the availabilities and characteristics of important system building blocks, including the available computation resources, memory resources, and communication interfaces on a given Xilinx FPGA device.

Component pre-characterization is involved in the modeling process. Specifically, it characterizes the delay, area, and power for each type of hardware resource, such as arithmetic units (e.g., adders and multipliers), memories (e.g., RAMs, ROMs, and registers), steering logic (multiplexors), and interface logic (e.g., FIFOs and bus interface adapters). The delay/area/power characteristic curves are derived by varying the bit widths, number of input and output ports, pipeline intervals, and latencies. The resulting characterization data is then used to make implementation choices during synthesis.

Notably, the cost of implementing hardware on FPGAs is often different from that for ASIC technology. For instance, most designs include multiplexors to route data to different points in a design, share hardware resources, and initialize the state of the system. On FPGAs, multiplexors typically have the same cost and delay as an adder (approximately one LUT/output). In some cases, however, a multiplexor can merge with other logic, such as a downstream adder or multiplexor, resulting in no additional hardware cost. In contrast, in ASIC technology, multiplexors are typically significantly less expensive than adders and other arithmetic operations and this cost cannot typically be eliminated by technology mapping. As a result, understanding the cost and delay of multiplexing operations is critical to building optimized FPGA designs.

FPGA technology also features heterogeneous on-chip resources, including not only LUTs and flip flops but also other prefabricated architecture blocks such as DSP48s and Block RAMs. Understanding the tradeoff between these heterogeneous resources is critical for efficient FPGA mapping. For instance, in FPGAs logic functions are significantly more expensive relative to memory than in ASIC technology, since logic functions must be implemented using LUTs and flip flops in the FPGA fabric whereas memory is usually implemented using Block RAMs which exist as customized SRAM cells in the FPGA. Furthermore, smaller memories and shift registers may be more efficiently mapped to LUT cells or flip flops in the FPGA than to Block RAM, adding additional complexity for memory characterization.

Such FPGA-specific platform information is carefully modeled for each and every FPGA device families, and considered by AutoPilot during synthesis for performance and area tradeoff. In addition, AutoPilot has the capability of detecting certain computation patterns and mapping a group of operations to platform-specific architecture blocks, such as DSP48 blocks, or pre-defined customer IPs.

### B. Integration with Xilinx toolset

In order to raise the level of design abstraction more completely, AutoPilot attempts to hide details of the downstream RTL flow from users as much as possible. Otherwise, a user may be overwhelmed by the details of vendor-specific tools such as the formats of constraint and configuration files, implementation and optimization options, or directory structure requirements.

As shown in Figure 1, AutoPilot implements an end-to-end C-to-FPGA synthesis flow integrated with the Xilinx toolset in several areas:

- **ISE integration**: AutoPilot automatically generates scripts and constraints for Xilinx ISE from the high-level constraints entered in AutoPilot. AutoPilot can also directly invoke ISE from within the tool to execute the entire C-to-FPGA flow and extract the exact resource utilization and the final timing from the ISE reports. For advanced users who are familiar with the Xilinx tool flow, AutoPilot also provides options to tune the default implementation and optimization settings, such as I/O buffer insertion, register duplication/balancing, and place-and-route effort.
- **CoreGen integration**: AutoPilot can automatically generate optimized IP blocks, such as memories, FIFOs, and floating-point units, using Xilinx Core Generator (CoreGen). In some cases, the CoreGen implementations are superior to the comparable functions implemented through logic synthesis

resulting in better QoR. The resulting CoreGen netlists are also incorporated and encapsulated without further user intervention.

- **EDK integration**: The hardware modules synthesized by AutoPilot can also be integrated into Xilinx EDK environment for system-level hardware/software co-design and exploration. Specifically, AutoPilot is capable of generating various bus interfaces, such as Xilinx Fast Serial Link (FSL) and Processor Local Bus (PLB) for integrating with MicroBlaze and PowerPC processors and Xilinx Native Port Interface (NPI) for integrating with external memory controllers. AutoPilot instantiates these interfaces along with adapter logic and appropriate EDK meta-information to enable a generated module can be quickly connected in an EDK system.

## VI. Advances in Synthesis and Optimization Algorithms

In this section we highlight some recent algorithmic advancement in HLS that we believe are important factors in improving the quality of results of the latest HLS tools and helping them to produce results that are competitive with manual designs.

### *A. Efficient mathematical programming formulations to scheduling*

Classical approaches to the scheduling problem in high-level synthesis use either conventional heuristics such as list scheduling [1] and force-directed scheduling [73], which often lead to sub-optimal solutions, due to the nature of local optimization methods, or exact formulations such as integer-linear programming [45], which can be difficult to scale to large designs. Recently, an efficient and scalable system of difference constraint (SDC) based linear-programming formulation for operation scheduling has been proposed [15]. Unlike previous approaches where using $O(m \times n)$ binary variables to encode a scheduling solution with $n$ operations and $m$ steps [45], SDC uses a continuous representation of time with only $O(n)$ variables: for each operation $i$, a scheduling variable $s_i$ is introduced to represent the time step at which the operation is scheduled. By limiting each constraint to integer-difference form, i.e.,

$$s_i - s_j \le d_{ij}$$

where $d_{ij}$ is an integer. It is shown that a totally unimodular constraint matrix can be obtained. A totally unimodular matrix defined as a matrix whose every square submatrix has a determinant of 0 or ±1. A linear program with a totally unimodular constraint matrix is guaranteed to have integral solutions. Thus, an optimal integer solution can be obtained without expensive branch-and-bound procedures.

Many commonly encountered constraints in high-level synthesis can be expressed in the form of integer-difference constraints. For example, data dependencies, control dependencies, relative timing in I/O protocols, clock frequencies, and latency upper-bounds can all be expressed precisely. Some other constraints, such as resource usage, cannot directly fit into the form. In such cases, approximations can be made to generate pair-wise which can then be expressed as integer-difference constraints. Other complex constraints can be handled in similar ways, using approximations or other heuristics. Thus, this technique provides a very flexible and versatile framework for various scheduling problems, and enables highly efficient solutions with polynomial time complexity.

### *B. Soft constraints and applications for platform-based optimization*

In a typical synthesis tool, design intentions are often expressed as constraints. While some of these constraints are essential for the design to function correctly, many others are not. For example, if the estimated propagation delay of a combinational path consisting of two functional units is 10.5 ns during scheduling, while the required cycle time is 10 ns, a simple method would forbid the two operations to execute in one clock cycle. However, it is possible that a solution with a slight nominal timing violation can still meet the frequency requirement, considering inaccuracy in interconnect delay estimation and various timing optimization procedures in later design stages, such as logic refactoring, retiming, and interconnect optimization. In this case, strict constraints eliminate the possibility of improving other aspects of the design with some reasonable estimated violations. In addition, inconsistencies in the constraint system can occur when many design intentions are added—after all, the design is often a process of making tradeoffs between conflicting objectives.

A solution to the above problems is proposed in [20] using soft constraints in the formulation of scheduling. The approach is based on the SDC formulation discussed in the preceding subsection, but allows some constraints to be violated. Consider the scheduling problem with both hard constraints and soft constraints formulated as follows.

| | | |
|---|---|---|
| *minimize* | $c^T s$ | linear objective |
| subject to | $\boldsymbol{G}s \le p$ | hard constraints |
| | $\boldsymbol{H}s \le q$ | soft constraints |

Here $\boldsymbol{G}$ and $\boldsymbol{H}$ corresponds to the matrices representing hard constraints and soft constraints, respectively, and they are both totally unimodular as shown in [15]. Let $\boldsymbol{H}_j$ be the $j$th row of $\boldsymbol{H}$, for each soft constraint $\boldsymbol{H}_j s \le q_j$, we introduce a violation variable $v_j$ to denote the amount of violation and transform the soft constraint into two hard constraints as

$$\boldsymbol{H}_j s - v_j \le q_j$$
$$-v_j \le 0$$

At the same time, we introduce a penalty term $\phi_j(v_j)$ to the objective function, to minimize the cost for violating the $j$th soft constraint. The final formulation becomes the following.

| | |
|---|---|
| *minimize* | $c^T s + \sum_j \phi_j(v_j)$ |
| *subject to* | $\boldsymbol{G}s \le p$ |
| | $\boldsymbol{H}s - v \le q$ |
| | $-v \le 0$ |

It can be shown that the new constraint matrix is also totally unimodular. If the amount of penalty is a convex function of

the amount of violation, the problem can be solved optimally within polynomial time. Otherwise, convex approximations can be made in an iterative manner [20].

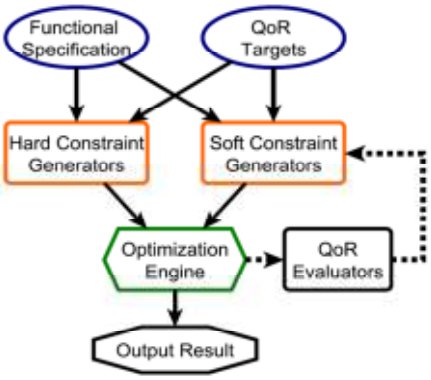


Figure 2. The structure of a scheduler using both hard constraints and soft constraints.

The overall flow of a scheduler using this method is shown in Figure 2. Hard constraints and soft constraints are generated based on the functional specification and QoR targets. The constraints are fed to an optimization engine that uses a mathematical programming solver. The soft constraints can be updated, based on existing results and possibly new design intentions. The use of soft constraints provides a way to handle multiple conflicting design intentions simultaneously, leading to efficient global optimization using a mathematical programming framework. This approach offers a powerful yet flexible framework to address various considerations in scheduling.

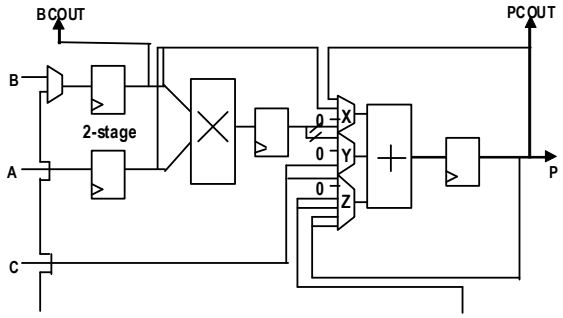


Figure 3. The Xilinx DSP48E block.

To illustrate the use of soft constraints in high-level synthesis for FPGAs, we apply it to the problem of efficient utilization of built-in fabrics on FPGA platforms. Take the DSP48E block in Xilinx Virtex 5 FPGAs for example: each of the DSP48E blocks (sketched in Figure 3) contains a multiplier and a post-adder, allowing efficient implementations of multiplication and multiply-accumulation. To fit the pattern of a DSP block, it is preferable that the operations are scheduled following certain relative cycle distances. Specifically, the addition should occur one cycle after the multiplication finishes to be mapped to the post-adder. In the constraint system, it is $s_+ - s_\times \le l_\times$, where $l_\times$ is the number of stages the multiplication takes. These preferences can be nicely modeled by soft constraints as they are not required for a correct implementation but highly preferred to achieve good QoR on FPGAs.

### C. *Pattern mining for efficient sharing*

A typical target architecture for HLS may introduce multiplexers when functional units, storage units or interconnects are shared by multiple operations/variables in a time-multiplexed manner. However, multiplexers (especially large ones) can be particularly expensive on FPGA platforms. Thus, careless decisions on resource sharing could introduce more overhead than benefit. In [16] a pattern-based approach for resource sharing is proposed. The method tries to extract common structures or patterns in the data-flow graph, so that different instances of the same pattern can share resources with little overhead. The approach tolerates small variations on port, bitwidth, operation types, etc., by using the graph editing distance as a metric to measure the similarity of two patterns. A systematic method for subgraph enumeration is developed which avoids generating redundant subgraphs. Pruning techniques are proposed based on characteristic vectors and locality-sensitive hashing. Instances of the same pattern are scheduled in the same way and conflicts are avoided when possible so that they can share resources, leading to resource reductions. This technique has been extended to pattern extraction and sharing in CDFGs [18].

### D. *Memory analysis and optimizations*

While application-specific computation platforms such as FPGAs typically have considerable computational capability, their performance is often limited by available communication or memory bandwidth. Typical FPGAs, such as the Xilinx Virtex series, have a considerable number of block RAMs. Using these RAMs effective is critical to meet performance target in many designs. This often requires partitioning elements of an array across multiple physical memory blocks to enable simultaneous access to different elements of the array.

In [19] a technique for automatic memory partitioning is proposed to increase throughput and reduce power for pipelined loops. It tightly integrates front-end transformations and operation scheduling in an iterative algorithm and has the ability to handle irregular array access, in addition to affine accesses. An example of memory partition is shown in Figure 4. Consider a loop that accesses array A with subscripts $i$, $2\times i+1$, and $3\times i+1$, in the $i$th iteration. When the array is partitioned into two banks, the first contains elements with even indices and the second contains those with odd indices. If the loop is targeted to be pipelined with the initiation interval of one, i.e., a new loop iteration starts every clock cycle, the schedule in (b) will lead to port conflicts, because $(i+1) \bmod 2 = (2\times(i+1)+1) \bmod 2 = (3\times i+1) \bmod 2$, when $i$ is even; this will lead to three simultaneous accesses to the first bank. On the other hand, the schedule in (c) can guarantee at most two simultaneous accesses. Because $(i+2) \bmod 2 \neq (3\times i+1) \bmod 2$ for any $i$, *R1* and *R3* will never access the same bank in the same cycle. The method in [19] presents a theorem to capture all possible reference conflicts under cyclic partitioning in a data structure called a *conflict graph*. Then, an iterative

algorithm is used to perform both scheduling and memory partitioning guided by the conflict graph.

```
for (i = 0; i < N; i++) {
    A[i] = …;           // R1
    A[2 * i + 1] = …;   // R2
    A[3 * i + 1] = …;   // R3
}
```

(a) The loop to be pipelined.

| | Step 1 | Step 2 | Step 3 |
|---|---|---|---|
| Iteration 1 | A[i]<br>A[2*i+1] | A[3*i+1] | |
| Iteration 2 | | A[i+1]<br>A[2*(i+1)+1] | A[3*(i+1)+1] |
| Iteration 3 | | | A[i+2]<br>A[2*(i+2)+1] |

(b) First schedule with partition.

| | Step 1 | Step 2 | Step 3 |
|---|---|---|---|
| Iteration 1 | A[i]<br>A[2*i+1] | | A[3*i+1] |
| Iteration 2 | | A[i+1]<br>A[2*(i+1)+1] | |
| Iteration 3 | | | A[i+2]<br>A[2*(i+2)+1] |

(c) Second schedule with partition.

Figure 4. An example of memory partitioning and scheduling for throughput optimization.

## VII. Advances in Simulation and Verification

Besides the many advantages of automated synthesis, such as quick design space exploration and automatic complex architectural changes like pipelining, resource sharing and scheduling, HLS also enables a more efficient debugging and verification flow at the higher abstraction levels. Since HLS provides an automatic path to implementable RTL from behavioral/functional models, designers do not have to wait until manual RTL models to be available to conduct verification. Instead, they can develop, debug and functionally verify a design at an earlier stage with high-level programming languages and tools. This can significantly reduce the verification effort due to the following reasons:

(i) It is easier to trace, identify and fix bugs at higher abstraction levels with more compact and readable design descriptions.

(ii) Simulation at the higher level is typically orders of magnitude faster than RTL simulation, allowing more comprehensive tests and greater coverage.

Figure 5 captures a typical simulation and verification framework offered by state-of-the-art C-based HLS tools. In this flow designers usually start from high-level specification in C/C++ or SystemC. They use software programming and debugging tools, such as GCC/GDB, Valgrind, or Visual Studio, to ensure that the design is sufficiently tested and verified against a properly constructed test bench. Once the input description to HLS is clean, designers can focus on the synthesis aspects and generate one or multiple versions of RTL code to explore the QoR tradeoffs under different performance, area, and power constraints. To confirm the correctness of the final RTL designers can use the automatic co-simulation and/or formal equivalence checking provided by this framework.

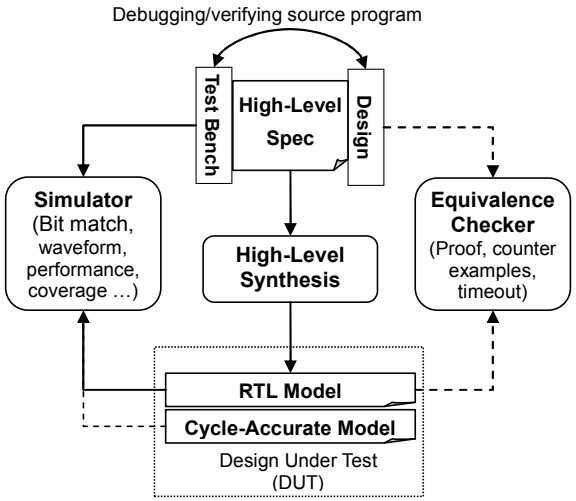


Figure 5. HLS simulation and verification framework.

### A. Automatic co-simulation

At present, simulation is the still prevalent technique to check if the resulting RTL complies with the high-level specification. To reduce effort spent on RTL simulation, the latest HLS technologies have made important improvements on automatic co-simulation [86][8][3], allowing direct reuse of the original test framework in C/C++ to verify the correctness of the synthesized RTL.

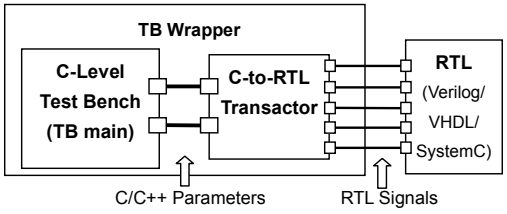


Figure 6. Automatic RTL test bench generation and connection in AutoPilot.

As an example, Figure 6 shows a block diagram describing how AutoPilot bridges a behavioral test bench (TB) and RTL with automatically constructed transactor and wrapper in SystemC. A C-to-RTL transactor is created to connect high-level interfacing constructs (such as parameters and global variables) with pin-level signals in RTL. This step involves data type synthesis as well as interface synthesis since the transactor needs to correctly translate various C/C++ data types and handle different interface protocols such as handshaking, streaming, and memory mapped I/O. Additionally, a SystemC wrapper is generated that combines the C-level test bench and transactor. This wrapper also includes additional control logic to manage the communication between the testing module and the RTL design under test (DUT). For instance, a pipelined design may require that the test bench feed input data into the DUT at a

fixed rate.

This style of automatic co-simulation also helps designers avoid the timing-consuming manual creation of an RTL test bench. Along with the use of instrumentation and code coverage tools, this flow can provide additional performance and code coverage analyses on the RTL output. Many HLS tools also generate alternative cycle-accurate models (typically in SystemC) of the synthesized design that can be more quickly simulated than HDL.

### *B. Equivalence Checking*

While formal equivalence checking tools for RTL-to-RTL and RTL-to-gate comparisons have been in production use for years, high-level to RTL checking is still an evolving technology.

Nevertheless, promising progress on C-to-RTL equivalence checking has been made in recent years, especially from industry. For instance, the Sequential Logic Equivalence Checker from Calypto [105] can identify mismatches between a synthesizable C/C++/SystemC model and an RTL design without the need of a test bench. This tool has been integrated in several commercial HLS flows. Synopsys has also presented their Hector tool in [53], which integrates multiple bit-level and word-level equivalence checking techniques, such as ATPG, BDD, SAT, and SMT to address the system level to RTL formal verification problem.

An excellent survey of the sequential equivalence checking (SEC) techniques is given in [63], with discussions of their usage in the real-word high-level synthesis flows. As mentioned in this article, the current SEC technology can handle moderate design size with gate count between 500-700K gates and tolerate latency differences between high-level and RTL models on the order of hundreds of clock cycles. Beyond this range, further design partitioning is required to help the checker to reduce the verification complexity.

Currently, formal equivalence checking plays a supporting role in the verification flow for HLS. This is particularly true for FPGA designs, where in-system simulation is possible with much wider simulation coverage. Design iterations can be performed quickly and inexpensively without huge manufacturing cost.

## VIII. Integration with Domain-Specific Design Platforms

In the end, the time-to-market of an FPGA system design is dependent on many factors, such as availability of reference designs, development boards, and in the end, FPGA devices themselves. Primarily, HLS only addresses one of these factors: the ability of a designer to capture new algorithms and implement an RTL architecture from the algorithm. Reducing the overall time-to-market requires not only reducing the design time, but also integrating the resulting design into a working system. This integration often includes a wide variety of system-level design concerns, including embedded software, system integration, and verification [104]. Hence, it is crucial that such integration can be performed as easily and as quickly as possible.

A view of an integrated design is shown in Figure 7. The interface cores (marked GigE, PCI, DVI, and LVDS in the figure) are implemented in low-level RTL code and are provided as encapsulated intellectual property (IP) cores. These cores tend to have tight requirements on circuit architecture in order to function correctly, and often have specific timing constraints, placement requirements, and instantiated architectural primitives. As a result, these cores are not easily amenable to high-level synthesis and form part of the system infrastructure of a design. Note, however, that these cores represent a small portion of the overall design synthesized in the FPGA, where system designers are not likely to have significant differentiating ability.

A second key part of system infrastructure is the processor subsystem shown on the left of Figure 7. Subsystem PSS is responsible for executing the relatively low-performance processing in the system.

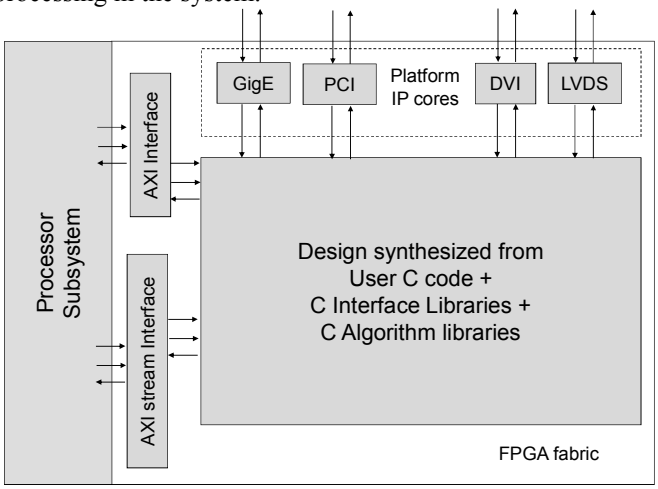


Figure 7. Block diagram showing an algorithmic block integrated with a processor and I/O.

The portion of a design generated using HLS represents the bulk of the FPGA design and communicates with the system infrastructure through standardized wire-level interfaces, such as AXI4 memory-mapped and streaming interfaces [96] shown in Figure 7. These interfaces are abstracted in the C code to appropriate application-level interfaces, which can be simulated at a functional level in C code. In order to understand this abstract architecture model, we show some concrete examples of domain-specific design platforms that we used to build FPGA systems, one for cognitive radio designs and another for video applications.

### *A. High-level design of cognitive radios project*

Cognitive radio systems typically contain both computationally intensive processing with high data rates in the radio processing, along with complex, but relatively low-rate processing to control the radio processing. Such systems can be elegantly described and quickly simulated in algorithmic C code, enabling opportunities to improve the system-level management algorithms. However, efficiently building such systems in FPGAs can be complex, since they

involve close interaction between the processing code that must be implemented in the FPGA fabricto provide adequate performance, and the control code that would typically be implemented in an embedded processor. Although HLS provides a path to implementing the radio processing efficiently in FPGA logic, efficient interaction with the processor is an important part of the overall system complexity.

The target template architecture, shown in Figure 8, is divided in two subsystems: a processor subsystem and an accelerator subsystem. The processor subsystem contains standard hardware modules and is capable of running a standard embedded operating system, such as Linux. These modules include the embedded CPU (e.g., PowerPC or MicroBlaze), memory controller to interface to external DRAM, and I/O modules (e.g., Ethernet). The processor subsystem is responsible for two main tasks: executing the software runtime system in charge of the application control at runtime, and executing computationally non-intensive components in the application. The accelerator subsystem is used for implementing components with high computational requirements in hardware. In order to transfer data into and out of the accelerator subsystem, the *accelerator* block is connected to on-chip memories (i.e., standard interfaces). These on-chip memories are used as a shared-memory communication scheme between hardware and software components. The bus interface logic implements a DMA functionality to efficiently move data. A set of interface registers, accessible from software, is used for controlling hardware execution and accessing component parameters. The accelerator block is synthesized using the high-level synthesis tools.

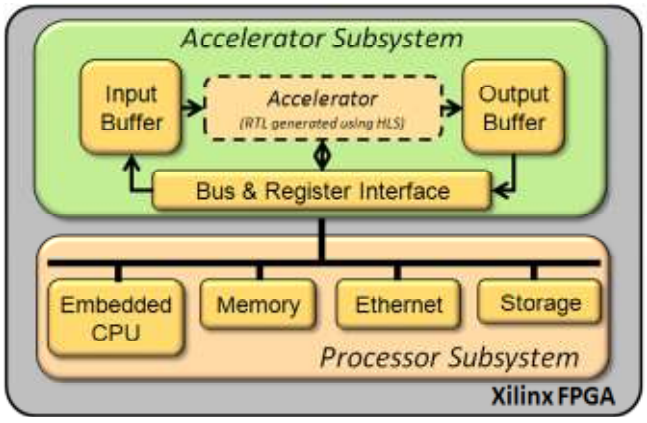


Figure 8. Radio processing architecture template.

To program the architecture, the application is captured as a pipeline of concurrent components or actors. Each actor conceptually executes either in the processor subsystem, or in the accelerator subsystem. Actors executing in the accelerator system also include a small proxy component executing in the processor, which is responsible for data transfer and synchronization with the FPGA hardware generated through HLS. This allows the component implementation to be completely abstracted, and a designer can implement individual components without knowing about the implementation details of other components or how they are logically interconnected. The composition of actors and the dataflow between them is described in an XML file, enabling new compositions to be easily described. Components also expose a configuration interface with multiple parameters, allowing them to be reconfigured in an executing system by user-defined control code executing in the processor subsystem.

### *B. Video Starter Kit*

Video processing systems implemented in FPGA include a wide variety of applications from embedded computer-vision and picture quality improvement to image and video compression. These systems also target a variety of end-markets ranging from television studio equipment to industrial imaging and consumer equipment, such as HDTVs and digital cameras. Typically these systems include two significant pieces of complexity. First, they must communicate by standardized interfaces, such as HD-SDI, HDMI, or V-by-one, with other equipment in order to be demonstrated. Secondly, they often perform inter-frame processing, which almost always requires a large frame-buffer implemented in cheap external memory, such as DDR2 SDRAM.

To address these complexities and make it relatively straightforward for designers to implement video processing applications and demonstrate them in real-time on development boards, we have leveraged a portable platform methodology. This platform is derived from the Xilinx EDK-based reference designs provided with the Xilinx Spartan 3ADSP Video Starter Kit and has been ported to several Xilinx Virtex 5 and Spartan 6 based development boards, targeting high-definition HD video processing with pixel clocks up to 150 MHz. A block diagram is shown in Figure 9.

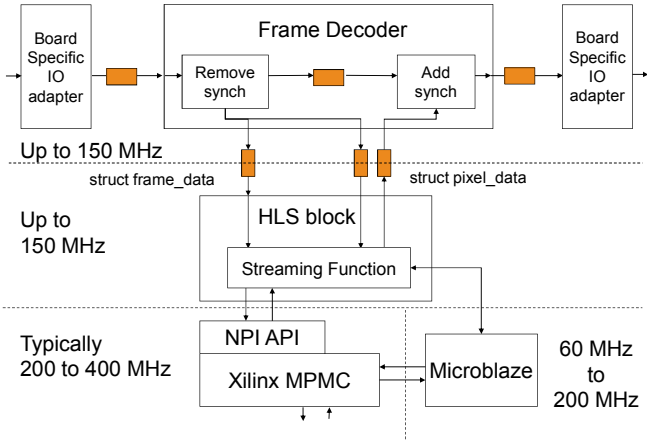


Figure 9. Video processing architecture template.

Incoming video data is received using board and protocol specific interface adapters and formatted as a non-handshaked stream of RGB video data, with horizontal and vertical synchronization and data enable signals. When a board uses an external decoder chip which formats digital video in this way, such as the Spartan 3ADSP video Starter Kit, the IO adapter can often be very simple, requiring almost no FPGA logic. In

other cases, such as on the Xilinx University Program Atlys board [117] which implements HDMI interfaces entirely in FPGA logic, the interface logic can be more significantly complex.

The incoming video data is analyzed by the Frame Decoder block to determine the frame size of the incoming video, which is passed to the application block, enabling different video formats to be processed. The frame size, represented by *struct frame_data*, is sent to the application block first, followed by the given number of active video pixels without synchronization signals, represented by *struct pixel_data*. The synchronization signals themselves are encoded and delayed, before being reassembled with the processed video data and sent to the output video interface. This delay accommodates non-causal spatial filters with up to a small number of lines of delay, without requiring the output video to be shifted. Longer processing delays can be accommodated internally to the application block by a frame buffer by outputting the previously processed frame.

The application is typically partitioned between the Application Block, which is generated using HLS and the Microblaze control processor. In video systems, the control processor often handles processing that occurs at the frame rate (typically 60 or 120 frames per second for the majority of consumer video equipment), and can receive data analyzed from the incoming video, and generate parameter updates to the processing core. Simple processing tasks can be computed in the vertical blanking interval, while more complex tasks may require the entire frame time to compute, meaning that analysis of frame n is computed during the arrival of frame $n$+1 and the results used to update frame $n$+2.

The Application Block itself is capable of processing video pixels at the full rate of the incoming video data, typically as a streaming dataflow pipeline generated from multiple loops in C code. To meet the pixel-rate requirements of HDTV systems, the Application Block typically process one new pixel per clock cycle in consumer grade FPGAs, such as the Xilinx Spartan 6 family. Video line buffers are synthesized directly from embedded FPGA memories, expressed as arrays in C code.

The interface to external memory used for frame buffers is implemented using the Xilinx Multi-ported Memory Controller (MPMC) [118] which provides access to external memory to the Application Block and to the Microblaze control processor, if necessary. The MPMC provides a consistent user-level interface through the Native-Port Interface (NPI) [118] to a variety of memory technologies, abstracting the FPGA-architecture specific details of interfacing with correct timing to a particular external memory technology. NPI requires applications to explicitly specify large bursts in order to maximize memory bandwidth to burst-oriented memory technologies, such as DDR2 SDRAM. The RTL code generated by AutoPilot can leverage these bursts to directly implement video frame buffers and other patterns of memory accesses without a separate DMA engine.

## IX. Design Experience and Results

In this section we summarize some recent design experiences using HLS for FPGA designs in the two application domains discussed in the preceding section and discuss the experimental results, especially in terms of the quality of results of HLS as compared to manual designs.

### *A. Summary of BDTI HLS Certification*

Xilinx has worked with BDTI Inc. [99] to implement an HLS Tool Certification Program [100]. This program was designed to compare the results of an HLS Tool and the Xilinx Spartan 3 FPGA that is part of the Video Starter Kit, with the result of a conventional DSP processor and with the results of a good manual RTL implementation. There were two applications used in this Certification Program, an optical flow algorithm, which is characteristic for a demanding image processing application and a wireless application (DQPSK) for which a very representative implementation in RTL was available. The results of the certification of the AutoPilot tool from AutoESL are available on the BDTI website [101].

Table 2. Quality of results for BDTI optical flow workflow operating point 2: maximum throughput, 1280x720 progressive scan. (Table reproduced from [101])

| Platform | Chip Unit Cost (Qty 10K) | Maximum Frames per Second (FPS) | Cost per FPS (Lower is Better) |
|---|---|---|---|
| AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3D3400A FPGA | $26.65 | 183 fps | $0.14 |
| Texas Instruments software development tools targeting TMS320DM6437 DSP processor | $21.25 | 5.1 fps | $4.20 |

Results showing the maximum performance for the optical flow algorithm are shown in Table 2, comparing comparably priced consumer-grade FPGA and DSP targets. The AutoPilot implementation achieved approximately 30 times better throughput per dollar than the optimized DSP implementation. In addition, BDTI qualitatively assessed the "extent of modifications to the source code" necessary to implement the optical flow algorithm. The DSP processor implementation rated "fair", while the AutoPilot implementation rated "good", indicating that less source code modification was necessary to achieve high performance when using AutoPilot.

Results for the DQPSK application are shown in Table 3, comparing the quality of results of the AutoPilot implementation with a manual RTL implementation. After optimization, including both significant source code refactoring and careful use of tool directives, the AutoPilot implementation achieved slightly lower resource usage than the RTL implementation. It is worth noting that the hand-written RTL made use of optimized Xilinx CoreGen IP blocks

where applicable.

BDTI also assessed overall ease of use of the DSP tool flow and the FPGA tool flow, combining HLS with the low-level implementation tools. They concluded that the DSP tool flow was still significantly easier to use, primarily due to difficulties installing the FPGA tools and a lack of sufficient platform infrastructure that can be accessed without in-depth knowledge of the FPGA tool flow. In the future, we believe that these issues will be solved as shown in Section VIII.

Table 3. Quality of results for DQPSK receiver workload: 18.75 MSamples/second input data at 75MHz clock speed. (Table reproduced from [101])

| Platform | Chip Resource Utilization (Lower is Better) |
|---|---|
| AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3D3400A FPGA | 5.6% |
| Hand-written RTL code using Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA | 5.9% |

### B. *Sphere Decoder*

Xilinx has implemented a sphere decoder for a multi-input multi-output (MIMO) wireless communication system using AutoPilot [67][85]. The algorithm [28] consists largely of moderate-throughput linear algebra operations, such as matrix-matrix multiply, matrix inverse, QR decomposition, and vector-norm computations implemented on small-dimension matrices. The application exhibits a large amount of parallelism, since the operations must be executed on each of 360 independent subcarriers which form the overall communication channel and the processing for each channel can generally be pipelined. However, in order to reach an efficient high-utilization design, the implementation makes extensive use of resource sharing and time-division multiplexing, with the goal of simultaneously reducing resource usage and end-to-end processing latency.

The algorithm was originally implemented in Matlab, which was converted to an algorithmic C model totaling approximately 4000 lines of code. The C model was further modified to generate an efficient implementation with AutoPilot. This code was converted to run through AutoPilot in a matter of days and optimized over a period of approximately three man-months. The resulting HLS code for the application makes heavy use of C++ templates to describe arbitrary-precision integer data types and parameterized code blocks used to process different matrix sizes at different points in the application. Various AutoPilot-specific *#pragma* directives were used, primarily to express the layout of arrays in memory blocks, to direct the unrolling and scheduling of loops to the appropriate level of parallelism, and to guide the scheduling algorithms to share operators and minimize resource usage. Most of the code included no explicit specification of the RTL structure, although in one case it was necessary to include a *#pragma* directive to force the RTL micro-architecture of a C function and to force the selection of a particular multiplier library element.

The end architecture consists of 25 independent actors in a streaming dataflow architecture, shown in Figure 10. Each actor is separated by synthesized streams or double buffers from neighboring components, enabling them to execute concurrently. The portions marked "4x4", "3x3" and "2x2" perform the same algorithm on matrices of decreasing size, and are collectively termed the "channel preprocessor". These portions are implementing using parameterized C++ templates, targeted by AutoPilot at different *II* [1] (3 in the 4x4 case, 5 in the 3x3 case and 9 in the 2x2 case), enabling optimized resource sharing decisions to be made automatically. The remainder of the design operates at *II*=1, with all resource sharing described in the C code.

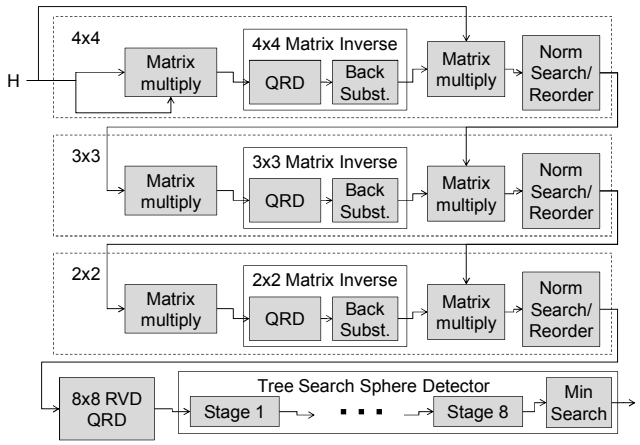


Figure 10. Architecture of the sphere decoder application.

Table 4 below summarizes the results, comparing the overall AutoPilot-based implementation with a previously reported RTL-style implementation built using Xilinx System Generator. Both designs were implemented as standalone cores using ISE 12.1, targeting Xilinx Virtex 5 speed grade 2 at 225 MHz. Using AutoPilot Version 2010.07.ft, we were able to generate a design that was smaller than the reference implementation in less time than a hand RTL implementation by refactoring and optimizing the algorithmic C model.

Design time for the RTL design was estimated from work logs by the original authors of [28], and includes only the time for an algorithm expert and experienced tool user to enter and verify the RTL architecture in System Generator. Design time for the AutoPilot design was extracted from source code control logs. It reflects the time taken by a tool expert who is not a domain expert to take a piece of unfamiliar code, implement a first version in the tool, refactor the code to reflect a desired target architecture, reverse engineer the original RTL code to discover that algorithmic improvements were made in the RTL implementation that were not reflected back in the algorithmic model, and perform design exploration.

[1] *II* denotes initiation interval of the pipeline. *II*=1 means the design accepts new inputs and produces new outputs at every clock cycle.

In both cases, multiple people worked on the design in parallel. Given the significant time familiarizing ourselves with the application and structure of the code, we believe that an application expert familiar with the code would be able to create such a design at least twice as fast.

Table 4. Sphere decoder implementation results.

| Metric | RTL expert | AutoPilot expert | Diff. (%) |
|---|---|---|---|
| Dev. time (man-weeks) | 16.5 | 15 | -9% |
| LUTs | 32,708 | 29,060 | -11% |
| Registers | 44,885 | 31,000 | -31% |
| DSP48s | 225 | 201 | -11% |
| 18K BRAMs | 128 | 99 | -26% |

To better understand the area savings, it is instructive to look more closely at smaller blocks of the design. The RVD-QRD block, summarized in Table 5, operates at *II*=1, completing an 8x8 QR decomposition of 18-bit fixed point values every 64 cycles. The block implements a standard Givens-rotation based systolic array consisting of diagonal and off-diagonal cells, where the diagonal cells compute an appropriate rotation, zeroing one of the matrix elements, and the off-diagonal cells apply this rotation to the other matrix elements in the same row. To meet the required throughput, one row of the systolic array is instantiated, consisting of one diagonal cell and 8 off-diagonal cells, and the remaining rows are time multiplexed over the single row. In addition, since the systolic array includes a recurrence, 15 channels are time-division multiplexed over the same hardware. Exactly the same architecture was implemented, although AutoPilot was able to generate a more optimized pipeline for the non-critical off-diagonal cell, resulting in slightly lower resource usage after optimization. After only 3 weeks, the AutoPilot design had met timing and throughput goals, but required more logic resources than the RTL design. After additional optimization and synthesis constraints on the DSP48 mapping, AutoPilot realized the same DSP48 mapping as the RTL design (3 DSP48s to implement the off-diagonal cell rotations and 6 DSP48s to implement the diagonal cell computation and rotation), including mapping onto the DSP48 post-adder.

Table 5. 8x8 RVD-QRD implementation results.

| Metric | RTL expert | AutoPilot expert | AutoPilot expert |
|---|---|---|---|
| Dev. time (man-weeks) | 4.5 | 3 | 5 |
| LUTs | 5,082 | 6,344 | 3,862 |
| Registers | 5,699 | 5,692 | 4,931 |
| DSP48s | 30 | 46 | 30 |
| 18K BRAMs | 19 | 19 | 19 |

Table 6 details multiple implementations of the "Matrix-Multiply Inverse" components, consisting of the combined Matrix Multiply, QR Decomposition, and Back Substitution blocks. This combination implements $(A^TA)^{-1}$ for various dimensions of 18-bit complex fixed-point matrices. In both RTL and AutoPilot design approaches, the 4x4 case was implemented first, and the 3x3 and 2x2 cases were derived from the 4x4 case. In RTL, resource sharing was implemented in a similar way for each case, with real and imaginary components time-multiplexed over a single datapath. Deriving and verifying the 3x3 and 2x2 case took approximately one week each. In AutoPilot, the three cases were implemented as C++ template functions, parameterized by the size of the matrix. All three cases were implemented concurrently, using a script to run multiple tool invocations in parallel. Depending on the matrix dimension, different initiation intervals were targeted, resulting in a variety of resource sharing architectures for each block, as shown in Figure 11.

Table 6. Matrix-Multiply Inverse implementation results.

| Metric | 4x4 RTL | 4x4 AP | 3x3 RTL | 3x3 AP | 2x2 RTL | 2x2 AP |
|---|---|---|---|---|---|---|
| Dev. Time (man-weeks) | 4 | 4 | 1 | 0 | 1 | 0 |
| LUTs | 9,016 | 7,997 | 6,969 | 5,028 | 5,108 | 3,858 |
| Registers | 11,028 | 7,516 | 8,092 | 4,229 | 5,609 | 3,441 |
| DSP48s | 57 | 48 | 44 | 32 | 31 | 24 |
| 18K BRAMs | 16 | 22 | 14 | 18 | 12 | 14 |

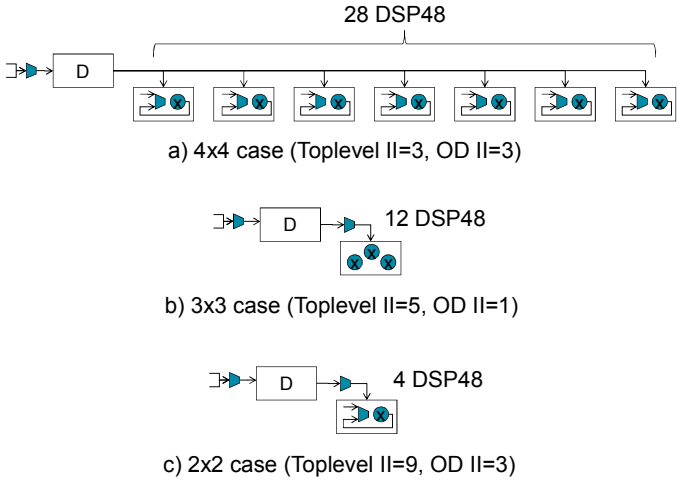


**Figure 11. Complex QRD architectures.**

In the 4x4 case, the off-diagonal cell implements fine-grained resource sharing, with one resource-shared complex multiplier. In the 3x3 case, the off-diagonal cell contains 3 complex multipliers and the off-diagonal cell itself is resource shared at a coarser granularity. In the 2x2 case, all of the off-diagonal cell operations are time multiplexed on a single complex multiplier, combining both coarse-grained and fine-grained resource sharing techniques. In AutoPilot, the only difference between these blocks is the different target initiation intervals, resulting in significant resource sharing. Certainly there is no doubt that an RTL designer could have achieved these architectures, given the appropriate insight. However, getting to the optimized cases from the 4x4 case implemented would require a complete RTL-level redesign. We do observe that AutoPilot uses additional BRAM to implement this block relative to the RTL implementation,

because AutoPilot requires tool-implemented double-buffers to only be read or written in a single loop. When considered as part of the overall design, however, we were able to reduce BRAM usage by converting BRAMs to LUTRAM due to the improve architecture of this block.

## X. Conclusions and Challenges ahead

It seems clear that the latest generation of FPGA HLS tools has made significant progress in providing wide language coverage, robust compilation technology, platform-based modeling, and domain-specific system-level integration. As a result, they can quickly provide highly competitive quality of results, in many cases comparable or better than manual RTL designs. For the FPGA design community, it appears that HLS technology may be transitioning from research and investigation to selected deployment.

Despite this encouraging development, we also see many opportunities for HLS tools to further improve. In this section, we discuss a few directions and opportunities.

### *A. Support of memory hierarchy*

The intelligent synthesis support of external off-chip memories is very important in applications that process large amounts of data or high data rates, for example:

- Data-intensive video and image processing applications often require multiple frames of data to be stored. In practice, this storage is usually implemented using commodity DDR2 SDRAMs and requires fast and efficient direct memory access logic to achieve high performance.
- Recent advances in FPGA-based high-performance reconfigurable computing [32] also require efficient access to the gigabytes of external memories shared between a host processor and an FPGA accelerator.

However, as mentioned in [66], most of the existing HLS solutions currently lack efficient support of the memory hierarchy and sufficient abstraction of the external memory accesses. As a result, software designers are exposed to the low-level details of bus interfaces and memory controllers. They must be intimately familiar with the bus bandwidth and burst length and translate such knowledge to C code with substantial modifications. Clearly, such design practice is out of the comfort zone for many software developers and algorithm designers.

Hence, it is highly preferable to have synthesis tools hide explicit external memory transfers as much as possible from programmers. This would require the support of efficient memory hierarchies, including automatic caching and prefetching to hide memory latency and enhance data locality.

The CHiMPS project [76] is one of the promising attempts in this area. It incorporates a traditional memory hierarchy with caches into the synthesized FPGA system to manage external memory, while focusing on the highest possible performance from a given code without rewriting. The proposed C-to-FPGA compilation flow generates multiple distributed caches used by multiple concurrent processing elements.

### *B. Higher-level models*

C and C++ languages are intended to describe sequential programs while modern FPGAs can implement highly complex concurrent systems. While the latest HLS tools have impressive capabilities to extract instruction-level and loop-level parallelism from C/C++ programs, it is inherently difficult to extract task-level parallelism from arbitrary sequential specifications. In addition, for systems with task-level feedback, sequential execution may not easily capture the parallel behavior of the system, making verification difficult.

Existing approaches mainly rely on manual annotation in the input specification for task-level parallelism. They also try to extract task-level parallelism by constructing synchronous data flow (SDF) [60], Kahn process networks (KPN) [48], or communicating sequential processes (CSP) [44] models from a sequential specification. A more effective approach may be to use programming models that can explicitly specify concurrency, dependency, and locality. For instance, recent work used the CUDA language [106] for input specification to HLS [68] since CUDA can easily describe thread-level concurrency. However, CUDA was originally intended to model applications mapped onto NVIDIA GPUs and includes many GPU specific features which are not suitable for FPGAs. Our preference is to choose a device-neutral programming model. Currently, we are investigating the possibility of using Concurrent Collections [108] to describe the task level dependency while continuing to specify each task using C/C++ languages.

### *C. In-System Design validation and debugging*

On-chip and on-board design validation and debugging has emerged as one of the most time-consuming aspects for FPGA-based systems, especially given continuously increasing device capacity and growing design complexity. Although the promise of HLS is that most verification can be performed by executing the original untimed C model, timing- and data-related errors that occur on the board are often difficult to debug. At present, the common practice to detect such errors is to perform RTL-level timing accurate simulation or to use in-system debugging tools from major vendors (e.g., Altera SignalTap II and Xilinx ChipScope). These tools can be used to insert logic analyzer cores and provide capabilities to trigger and probe internal signals inside the FPGA circuits.

Debugging HLS designs at the RTL level is complicated by the fact that the structure of the original C code may not resemble the RTL architecture generated by an HLS tool. Many of the modern HLS solutions provide cross referencing capabilities between C and RTL to help designers understand the synthesis results. However, names in HDL are often transformed during RTL synthesis and technology mapping.

In order to effectively debug these systems, future HLS tools shall enable almost all debugging to occur in the C domain by providing:

- Debugging core synthesis: the ability to synthesize efficient debugging logic with minimal overhead.
- Performance monitor generation: the ability to watch the

status of critical buffers to debug performance bugs, such as FIFO overflows and deadlocks.

- Step-through tracing: the ability to set breakpoints at the C level and observe internal states from hardware blocks.

## ACKNOWLEDGMENT

This work is partially supported by the Gigascale System Research Center (GSRC) and Global Research Corporation (GRC). Sven van Haastregt contributed significantly to the implementation and analysis of the sphere decoder, and his efforts are highly appreciated.

## REFERENCES

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17(12), pp. 685-690, Dec. 1974.

[2] B. Alpern, W. Bowen, M.N. Wegman, and F.K. Zadeck, "Detecting equality of variables in programs," in *Proc. PPL'88*, pp. 1-11.

[3] B. Bailey, F. Balarin, M. McNamara, G. Mosenson, M. Stellfox, and Y. Watanabe, "TLM-Driven Design and Verification Methodology," Cadence Design Systems, Jun. 2010.

[4] M. Barbacci, G. Barnes, R. Cattell, and D. Siewiorek, The symbolic manipulation of computer descriptions: the ISPS computer description language, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, *PA,Tech. Rep.*, Mar. 1978.

[5] R.A. Bergamaschi, R.A. O'Connor, L. Stok, M.Z. Moricz, S. Prakash, A. Kuehlmann and D. S. Rao, "High-level synthesis in an industrial environment," *IBM Journal of Research and Development*, vol. 39(1-2), pp. 131-148, Jan. 1995.

[6] J. Biesenack, M. Koster, A. Langmaier, S. Ledeux, S. Marz, M. Payer, M. Pilsl, S. Rumler, H. Soukup, N. Wehn, and P. Duzy, "The Siemens high-level synthesis system CALLAS," *IEEE Trans. VLSI Systems*, vol.1(3), pp.244-253, Sep 1993.

[7] T. Bollaert, "Catapult synthesis: a practical introduction to interactive C synthesis," in P. Coussy and A. Morawiec Eds. *High-Level Synthesis: From Algorithm to Digital Circuit*, Springer, 2008.

[8] D. Burnette, "An ESL methodology for functional verification between untimed C++ and RTL using SystemC," Mentor Graphics, 2008.

[9] M. Budiu, G. Venkataramani, T. Chelcea and S. Goldstein, "Spatial computation," in *Proc. ASPLOS'04*, pp. 14-26.

[10] A. Chandrakasan, M. Potkonjak, J. Rabaey, and R. Brodersen, "HYPER-LP: a system for power minimization using architectural transformations," in *Proc. ICCAD'92*, pp. 300-303.

[11] R. Composano, "Design process model in the Yorktown Silicon Compiler," in *Proc. DAC'88*, pp. 489-494.

[12] R. Composano, "Path-based scheduling for synthesis," *IEEE Trans. CAD*, vol. 10(1), pp. 85-93, Jan. 1991.

[13] J. Cong and Y. Zou, "Lithographic aerial image simulation with FPGA-based hardware acceleration," in *Proc. FPGA'08*, Feb. 2008, pp. 20-29.

[14] J. Cong and W. Rosenstiel, "The last byte: the HLS tipping point," *IEEE Design & Test of Computers*, vol. 26(4), pp. 104, Jul./Aug. 2009.

[15] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proc. DAC'06*, pp. 433-438.

[16] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction," in *Proc. FPGA'08*, Feb. 2008, pp. 107-116.

[17] J. Cong, Y. Fan, G. Han, W. Jiang and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *Proc. IEEE Intl. SOC Conf.*, Sept. 2006, pp. 199-202.

[18] J. Cong, H. Huang and W. Jiang, "A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis", in *Proc. DATE'10*, pp. 1255-1260.

[19] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," in *Proc. ICCAD '09*, Nov. 2009, pp. 697-704.

[20] J. Cong, B. Liu, and Z. Zhang, "Scheduling with soft constraints," in *Proc. ICCAD'09*, Nov. 2009, pp. 47-54.

[21] J. Cong, K. Gururaj, B. Liu, C. Liu, Z. Zhang, S. Zhou and Y. Zou, "Evaluation of static analysis techniques for fixed-point precision optimization," in *Proc. FCCM'09*, Apr. 2009, pp. 231-234.

[22] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: A High-Level Synthesis Tool for DSP Applications," in P. Coussy and A. Morawiec Eds. High-Level Synthesis: From Algorithm to Digital Circuit, Springer, 2008..

[23] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13(4), pp. 451-490, Oct. 1991.

[24] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus synthesis system," *IEEE Design & Test of Computers*, vol. 7(5), pp. 37-53, 1990.

[25] G. De Micheli and D. Ku, "HERCULES—A system for high-level synthesis," in *Proc. DAC'88*, pp. 483-488.

[26] H. De Man, J. Rabaey, J. Vanhoof, P. Six, and L. Claesen, "Cathedral-II—a silicon compiler for digital signal processing," *IEEE Design &Test of Computers*, vol. 3(6), pp. 13-25, 1986.

[27] K. Denolf, S. Neuendorffer, and K. Vissers, "Using C-to-gates to program streaming image processing kernels efficiently on FPGAs," in *Proc. FPL'09*, pp. 626-630.

[28] C. Dick et.al., "FPGA Implementation of a Near-ML Sphere Detector for 802.16e Broadband Wireless Systems," in *Proc. Of the SDR'09 Technical Conference*, Dec. 2009.

[29] S. Director, A. Parker, D. Siewiorek, and D. Thomas Jr."A design methodology and computer aids for digital VLSI," *IEEE Trans. Circuits and Systems*, vol. CAS-28(7), pp. 634-645, Jul. 1982.

[30] S. A. Edwards, "High-level synthesis from the synchronous language Esterel," in *Proc. IWLS'02*.

[31] S. A. Edwards. "The challenges of synthesizing hardware from C-like Languages," *IEEE Design & Test of Computers*, vol. 23(5), pp. 375-386, Sept. 2006.

[32] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *IEEE Computer*, vol. 41(2), pp. 69-76, Feb. 2008.

[33] J. P. Elliott, "Understanding behavioral synthesis: a practical guide to high-level design," Springer, 1999.

[34] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer and S. Zhao, *SpecC: specification language and methodology*, Kluwer Academic Publishers, 2000.

[35] F. Ghenassia, Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems, Springer, 2005.

[36] M. Gokhale, J. Stone, J. Arnold, M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *Proc. FCCM'00*, pp.49-56.

[37] J. Granacki, D. Knapp, and A. Parker, "The ADAM advanced design automation system: overview, planner and natural language interface," in *Proc. DAC '85*, pp. 727-730.

[38] Y. Guo, D. McCain, J. R. Cavallaro, and A. Takach, "Rapid industrial prototyping and SoC design of 3G/4G wireless systems using an HLS methodology," *EURASIP Journal on Embedded Systems*, vol. 2006(1), 2006.

[39] Z. Guo, B. Buyukkurt, J. Cortes, A. Mitra, and W. Najjar, "A compiler intermediate representation for reconfigurable fabrics," *International Journal of Parallel Programming*, vol. 36(5), pp. 493-520, Oct. 2008.

[40] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Trans. Design Automation of Electronic Systems*, vol. 9(4), pp. 441–470, Oct. 2004.

[41] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, "SPARK: a parallelizing approach to the high-level synthesis of digital circuits," Springer, 2004.

[42] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, "A system for synthesizing optimized FPGA hardware from MATLAB," in *Proc. ICCAD'01*, pp. 314-319.

[43] A. Hemani, B. Karlsson, M. Fredriksson, K. Nordqvist, and B. Fjellborg, "Application of high-level synthesis in an industrial project," in *Proc. VLSI Design'94*, pp.5-10.

[44] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21(8), pp. 666-677, Aug. 1978.

[45] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high-level synthesis," *IEEE Trans. Computer-Aided Design*, vol. 10(4), pp. 464-475, Apr. 1991.

[46] R. Jain, K. Kucukcakar, M.J. Mlinar, and A.C. Parker, "Experience with ADAM synthesis system," in Proc. DAC '89, pp.56-61.

[47] R. Jain, A. Parker, and N. Park, "Module selection for pipelined synthesis," in *Proc. DAC'88*, pp. 542-547.

[48] G. Kahn, "The semantics of a simple language for parallel programming," In *Jack L. Rosenfeld (Ed.): Information Processing 74, Proceedings of IFIP Congress 74*, Aug. 1974.

[49] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. C. Cronquist, and M. Sivaraman, "PICO: automatically designing custom computers," *IEEE Computer*, vol. 35(9), pp. 39-47, Sep. 2002.

[50] K. Kennedy and J.R. Allen, "Optimizing compilers for modern architectures: a dependence-based approach," Morgan Kaufmann Publishers, 2001.

[51] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: orthogonalization of concerns and platform-based design," *IEEE Trans. CAD*, vol. 19(12), pp. 1523-1543, Dec. 2000.

[52] V. Kindratenko and R. Brunner, "Accelerating cosmological data analysis with FPGAs", In *Proc. FCCM'09*, pp. 11-18.

[53] A. Kölbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to RTL equivalence checking," in *Proc. DATE 2009*, pp. 196-201.

[54] D. Ku and G. De Micheli, "HardwareC—a language for hardware design (version 2.0)," *Technical Report. UMI Order Number: CSL-TR-90-419*, Stanford University, 1990.

[55] D. Ku and G. De Mecheli, "Constrained resource sharing and conflict resolution in Hebe," *Integration, the VLSI Journal*, vol. 12(2), pp. 131-165, 1991.

[56] D. Ku and G. De Micheli, "Relative scheduling under timing constraints," in *Proc. DAC'91*, pp. 59-64.

[57] F. J. Kurdahi and A. C. Parker, "REAL: a program for register allocation," in *Proc. DAC'87*, pp. 210-215.

[58] K. Küçükçakar, C.-T. Chen, J. Gong, W. Philipsen and T. E. Tkacik, "Matisse: an architectural design tool for commodity ICs," *IEEE Design and Test of Computers*, vol. 15(2), pp. 22-33, Apr.-June 1998.

[59] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proc. CGO'04*, pp. 75-86.

[60] E. A. Lee and David G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE,* vol. 75(9), pp. 1235-1245, Sept. 1987.

[61] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney, J.O. Huisken and O.P. McArdle, "PHIDEO: a silicon compiler for high speed algorithms," in *Proc. EDAC'91*, pp.436-441.

[62] P. Marwedel, "The MIMOLA design system: tools for the design of digital processors," in *Proc.* DAC'84, pp.587-593.

[63] A. Mathur, M. Fujita, E. Clarke, and P. Urard, "Functional equivalence verification tools in high-level synthesis flows," *IEEE Design & Test of Computers*, vol. 26(4), pp. 88-95, Dec. 2009.

[64] O. Mencer, "ASC: a stream compiler for computing with FPGAs," *IEEE Trans. CAD*, vol. 25(9), pp. 1603-1617.

[65] M. Meredith, "High-level SystemC synthesis with Forte's Cynthesizer," in P. Coussy and A. Morawiec Eds. High-Level Synthesis: From Algorithm to Digital Circuit, Springer, 2008.

[66] S. Neuendorffer and K. Vissers, "Streaming systems in FPGAS," in SAMOS Workshop, ser. *Lecture Notes in Computer Science*, no. 5114, pp. 147-156, Jul. 2008.

[67] J. Noguera, S. Neuendorffer, S. Van Haastregt, J. Barba, K. Vissers, and C. Dick, "Sphere detector for 802.16e broadband wireless systems implementation on FPGAs using high-level synthesis tools," in *SDR'10 Forum,* Nov. 2010.

[68] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.M. Hwu, "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs," in *Proc. IEEE Symposium on Application Specific Processors*, Jul. 2009.

[69] N. Park and A. Parker, "Sehwa: a program for synthesis of pipelines," in Proc. DAC'86, pp. 595-601.

[70] A. Parker, J. T. Pizarro, and M. Mlinar, "MAHA: a program for datapath synthesis," in *Proc. DAC'86*, pp. 461-466.

[71] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim, "The CMU design automation system: an example of automated data path design," in *Proc. DAC'79*, pp. 73-80.

[72] P. G. Paulin. J. P. Knight, and E.F. Girczyc, "HAL: A multi-paradigm approach to automatic data path synthesis," in *Proc. DAC'86*, pp. 263-270.

[73] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8(6), pp. 661-678, Jun. 1989.

[74] D. Pellerin and S. Thibault, "Practical FPGA programming in C," Prentice Hall Professional Technical Reference, 2005.

[75] P.J. Pingree, L.J. Scharenbroich, T.A. Werne, and C.M. Hartzell, "Implementing legacy-C algorithms in FPGA co-processors for performance accelerated smart payloads," in *Proc. IEEE Aerospace Conference*, Mar. 2008, pp. 1-8.

[76] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig, "Performance and power of cache-based reconfigurable computing," in *Proc. ISCA'09*, pp. 395-405.

[77] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architectures," *IEEE Design & Test*, vol. 8(2), pp. 40-51, Apr. 1991.

[78] J. Sanguinetti, "A different view: hardware synthesis from SystemC is a maturing technology," *IEEE Design & Test of Computers*, vol. 23(5), pp. 387-387, Sept. 2006.

[79] E. Snow, D. Siewiorek, and D. Thomas, "A technology-relative computer aided design system: abstract representations, transformations, and design tradeoffs," in *Proc. DAC'78*, pp. 220-226.

[80] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation," in *Proc. PLDI'00*, pp. 108-120.

[81] Y. Sun, J.R. Cavallaro, and T. Ly, "Scalable and low power LDPC decoder design using high level algorithmic synthesis," in *Proc. IEEE SOC Conference*, 2009, pp. 267-270.

[82] J. L. Tripp, M. B. Gokhale, and K.D. Peterson, "Trident: from high-level language to hardware circuitry," *IEEE Computer*, vol. 40(3), pp. 28-37, Mar. 2007.

[83] J. L. Tripp, K.D. Peterson, C. Ahrens, J. D. Poznanovic and M.B. Gokhale, "Trident: an FPGA compiler framework for floating-point algorithms," in *Proc. FPL'05*, 2005, pp.317-322.

[84] P. Urard, J. Yi, H. Kwon and A. Gouraud, "User needs," in P. Coussy and A. Morawiec Eds. High-Level Synthesis: From Algorithm to Digital Circuit, Springer, 2008.

[85] S. van Haastregt, S. Neuendorffer, K. Vissers, and B. Kienhuis, "High level synthesis for FPGAs applied to a sphere decoder channel preprocessor," Manuscript.

[86] D. Varma, D. Mackay, and P. Thiruchelvam, "Easing the verification bottleneck using high level synthesis," *IEEE VLSI Test Symposium*, Apr. 2010.

[87] J. Villarreal, A. Park, W. Najjar and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proc. FCCM'10*, pp.127-134.

[88] D. W. Knapp, "Behavioral synthesis: digital system design using the Synopsys Behavioral Compiler," Prentice-Hall, 1996.

[89] K. Wakabayashi and B. Schafer, "'All-in-C' behavioral synthesis and verification with CyberWorkBench," in P. Coussy and A. Morawiec Eds. High-Level Synthesis: From Algorithm to Digital Circuit, Springer, 2008.

[90] K. Wakabayashi, "C-based behavioral synthesis and verification analysis on industrial design examples," in *Proc. ASPDAC'04*, pp. 344-348.

[91] K. Wakabayashi, "Unified representation for speculative scheduling: generalized condition vector," *IEICE Transactions*, vol. E89-A(12), pp. 3408-3415, 2006.

[92] F. F. Yassa, J. R. Jasica, R. I. Hartley, and S. E. Noujaim, "A silicon compiler for digital signal processing: methodology, implementation, and applications," in *Proc. IEEE*, vol. 7(9), pp. 1272-1282.

[93] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong, "Bit-level optimization for high-level synthesis and FPGA-based acceleration," in *Proc. FPGA'10*, Feb. 2010, pp. 59-68.

[94] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "*AutoPilot: a platform-based ESL synthesis system*," High-Level Synthesis: From Algorithm to Digital Circuit, ed. P. Coussy and A. Morawiec, Springer Publishers, 2008.

[95] Agility Design Solutions, *Handel-C language reference manual*, 2007.

[96] AMBA AXI specification, version 2.0. http://www.arm.com.

[97] Avnet Spartan-6 FPGA DSP Kit. http://www.xilinx.com/products/devkits/AES-S6DSP-LX150T-G.htm.

[98] Altera Corporation, *Nios II C2H compiler user guide*, *version 9.1*, Nov. 2009.

[99] Berkeley Design Technology, Inc. (BDTI), www.bdti.com.

[100] BDTI High-Level Synthesis Tool Certification Program™, http://www.bdti.com/products/services_hlstcp.html, Jan. 2010.

[101] BDTI Certified™ Results for the AutoESL AutoPilot High-Level Synthesis Tool, http://www.bdti.com/bdtimark/hlstcp_autoesl.html, Jan. 2010.

[102] BlueSpec, Inc. http://www.bluespec.com.

[103] Cadence C-to-Silicon white paper. http://www.cadence.com/rl/resources/technical_papers/c_to_silicon_tp.pdf, 2008.
[104] Cadence EDA360 Whitepaper. Silicon realization enables next-generation IC design.
[105] Calypto Design Systems, Inc., http://www.calypto.com.
[106] Compute Unified Device Architecture Programming Guide. NVIDIA, 2007.
[107] IEEE and OCSI. IEEE 1666™-2005 Standard for SystemC. http://www.systemc.org, 2005.
[108] Intel Concurrent Collections for C++, http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc, Apr. 2010.
[109] International Technology Roadmap for Semiconductors (ITRS), 2009 edition. http://www.itrs.net/links/2009ITRS/Home2009.htm.
[110] LLVM compiler infrastructure. http://www.llvm.org.
[111] Magma Talus. http://www.magma-da.com/products-solutions/digitalimplementation/talusdesign.aspx
[112] Nallatech, Inc., *DIME-C user guide*.
[113] Open SystemC Initiative. SystemC Synthesizable Subset Draft 1.3. http://www.systemc.org, Dec. 2009.
[114] Synopsys IC compiler. http://www.synopsys.com/Tools/Implementation/PhysicalImplementation/Pages/ICCompiler.aspx.
[115] Synopsys Synphony. http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx
[116] Xilinx Spartan-6 FPGA Consumer Video Kit. www.xilinx.com/products/devkits/TB-6S-CVK.htm.
[117] Xilinx University Program. http://www.xilinx.com/products/boards_kits/university.
[118] Xilinx Multi-ported Memory Controller (MPMC) Data Sheet (v6.02.a), DS643, Sep. 21, 2010.

Professor Vijaykrishnan Narayanan
Computer Science and Engineering
The Pennsylvania State University University Park, PA 16802

Dear Prof. Narayanan:

Thank you very much for sending us the reviewers' comments on our paper entitled "High-Level Synthesis for FPGAs: From Prototyping to Deployment" by Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang (Control Number: 6158). We found these comments very helpful. We have revised the manuscript carefully according to reviewers' comments and suggestions. Here is a brief summary of our major revisions:

(i) More data regarding design experience and results are included. This gives more details on the use of high-level synthesis tools in an industrial setting, and clarifies possible misunderstandings about the efficiency of high-level synthesis tools.
(ii) The section that reviews previous and concurrent high-level synthesis tools is substantially rewritten. Instead of describing features of each tool individually, we grouped them according to their contributions and timelines, with an emphasis to show the progression of the HLS technology. We then present AutoPilot in this backdrop, describing it as one of the representative tools in the current market, and avoid direct comparison between AutoPilot and other prevalent tools.
(iii) A new section is added to describe debugging, simulation and verification in high-level synthesis per reviewers' request.

We also made numerous minor changes based on reviewers' comments, which are detailed in the attachment. We would appreciate it if you would forward this letter to the associate editor and the reviewers. Thank you very much for your help.

Sincerely,

Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang
AutoESL Design Technologies, Inc.
University of California, Los Angeles
Xilinx, Inc.

Responses to Comments from Reviewer Number 1

(1) *Being a relevant competitor to AutoESL I would have, however, expected to see more distinction between the Mentor Graphic Catapult-C tool and AutoPilot, especially in the results section.*
Reply: Thanks for the suggestion. We agree that it would be useful for readers to see a comparison between Catapult-C and AutoPilot. However, there is difficulty to carry out such a comparison for multiple reasons: (i) we do not have access to Catapult-C, and (ii) Almost all CAD tool end-user license agreements have confidentiality clauses that prohibit direct comparison of the tool's performance with other competing tools. Such a comparison is only possible when coordinated by a third-party organization with explicit consent of CAD tool vendors. Obviously, this is beyond the effort of the paper. We quoted the results from the recent BDTi a high-level synthesis tool certification program. AutoPilot was one of the tools evaluated for quality of result and for usability. Unfortunately, Mentor Catapult-C did not participate in this program. We are definitely open to discuss the comparison if similar programs are held in future.

(2) *Most importantly, Table 2, shows a 1.5 week advantage in development time for an expert AutoPilot user. In my opinion this is not a substantial reduction in time to convince an expert RTL designer to adopt this flow.*
Reply: Thanks for pointing it out.
We were trying to give a comparison between the AutoPilot flow and the sysgen-RTL flow by reporting the time to capture and verify the design, as extracted from source-code control logs. In both cases, a process of converting the reference model in Matlab is included. Yet, the RTL design was performed by a team including someone who are very familiar with the Matlab model, and the 16.5 weeks of design time does not include that spent on learning about the algorithm. On the other hand, the team using HLS flow did not have the domain knowledge initially, and thus spent a significant amount of time understanding the Matlab code and converting it to C++. We believe that a domain expert would be able to create the C++ model much faster. We have clarified this in the revised manuscript (Page 14, last paragraph).
Once a reference C/C++ model is available, it is very easy to obtain RTL results using AutoPilot. It is very easy to explore the design space by changing synthesis directives and constraints. This fast exploration actually provided us a way to get better QoR. Using a recent version of AutoPilot, we are actually able to get significant QoR improvements (see Table 4, 5, 6). In Table 6 of the revised manuscript, we report different architectures for the matrix-multiply inverse module. Once the 4x4 case is implemented in C, obtaining the 3x3 and 2x2 cases took almost no time. We believe these results are encouraging for the deployment of HLS tools.

(3) *Finally, the validation and debugging limitations of HLS are of large importance deserving of more discussion than that in section IX.C.*
Reply: Thanks for the suggestion. In the revised manuscript, we have Section VII devoted to design verification. We included both simulation-based approach and the formal approach.

Responses to Comments from Reviewer Number 2

(1) *For the second point of section II.C, you have said the existing hls tools required users to embed detailed timing and interface information as well as the synthesis constraints into the source code. Please explain it or cite some references.*
Reply: When synthesizing a module described in behavior languages like C/C++ into RTL, it is important that the module meets the timing specifications, including interface protocol, latency, etc. A typical approach to specify these constraints in C/C++ file is to use compiler pragmas in the source code. However, different tools often require different formats to impose the constraints, making the specification not portable.

(2) *What's the benefit of using soft constraints?*
Reply: Using soft constraints is a natural way to specify non-functional design requirements or intentions. Soft constraints do not lead to infeasible constraint systems and thus can be imposed without worrying consistency with other existing constraints. In addition, the special form of soft constraints (integer-difference soft constraints) can be handled optimally in a mathematical-programming formulation. In the revised manuscript, we have included the formulations and added more explanation on this part (Section VI.B).

(3) *Please make fig 7 clearer, some words in fig 7 cannot be seen clearly.*
Reply: Thanks for the suggestion. We have included a clearer version of the figure. We have also redrawn several other figures are redrawn in vector graphic format in the revised manuscript.

Responses to Comments from Reviewer Number 3

(1) *The paper does provide a valuable survey of HLS history but then recent achievements in HLS are restricted to the authors own work. Why not include a survey of recent algorithm advancements by other researchers as well to help make the case that HLS has advanced? The paper currently makes it seem like the only valuable recent HLS research in the past five to ten years has been on AutoPilot.*

*Inclusion of a number of recent C-language HLS advancements that were not made by the authors. Published work by Impulse-C and Celoxia should be considered and included in a somewhat significant way in the main body of the paper, not just in the background section. The authors should at least mention that Altera has a C2H tool.*
Reply: Thanks for your suggestion. We have substantially reorganized Section II, and added a lot of new materials. More tools (including Impulse-C, Celoxica, and Altera C2H) are included, and they are categorized in various aspects to provide more insight.

(2) *Significantly more results (e.g. at least 3 or 4 designs) showing that AutoPilot can generate code for a Xilinx device that is close to hand-written RTL are needed.*
Reply: Thanks for your suggestion. We have expanded the section about design experience and quality-of-results, with more modules and QoR data. Table 4-6 show that AutoPilot is capable of producing comparable or better QoR than manual RTL on industrial designs.

(3) *Removal of various marketing type statements about how designers prefer AutoPilot, its easy-of-use, its acceptance in the marketplace, how it addresses failures of other company's products, etc. In my opinion these types of statements belong in trade magazines, not top research journals like TCAD*
Reply: In the revised manuscript, we removed such statements and avoided direct comparison between AutoPilot and other tools in the market. Instead, we position AutoPilot as one of the representative C-base synthesis tools and try to describe the current status of HLS adoption using AutoPilot as an example.

(4) *The paper is a little long and could be shorted. Some of the background material on the early HLS systems is verbose and could be trimmed. There could be more of a focus on the specific contributions of each tool rather than details of all their features. The "Lessons Learned" section could be merged with the "Early Efforts" section to make the deficiencies of earlier systems clearer (e.g. which specific system had which deficiency?). At a minimum, the "Lessons Learned" section should include references to make it clearer which previous systems had these deficiencies. A lot of Sections V.A and V.B is repeated in Section VII.*
Reply: We have reorganized Section II in the revised manuscript to make it more concise and insightful, by categorizing tools according to different aspects. We have a “lessons learned” subsection without specifying which tool has which deficiency for several reasons. (1) Some of the deficiencies are shared by many tools, and we have tried to discuss the pros and cons for the general approach (such as “using C/C++ as input language”). (2) With some of the authors being affiliated with AutoESL, emphasizing the deficiency of tools from our competitors seems inappropriate in a technical article, and thus is avoided. (3) Many of the tools are still evolving; a specific deficiency of a tool may be fixed in the near future (or maybe have already been fixed at the time when the paper is published).

(5) *I'm not sure why Figure 3 is included.*
Reply: Thanks for pointing out the missing reference. We have fixed the problem as well as a few other similar ones in the revised manuscript.

Responses to Comments from Reviewer Number 4

(1) *In some figures (e.g. 7) you need to increase the size of the captures or lines and maybe you should not use colors. Greyscale is better since many often, the papers are printed black and white.*
Reply: Thanks for your suggestion.
We have edited the figures to make them more visible in the revised manuscript.

Responses to Comments from Reviewer Number 5

(1) *I have included the PDF file that is marked with small grammatical edits.*
Reply: Thank you very much for your help. We have corrected all the mistakes you pointed out, and refined the language/grammar for the entire article.

(2) *The biggest problem I have with this paper is the design validation and debugging at the end of the paper. It should be moved forward to section V or so.*
*In general, debugging with HLS is not addressed until the end of the paper and should be addressed before that. It is hard to debug FPGAs when using VHDL descriptions, using HLS is even harder because of the mapping between the VHDL and high-level description.*
Reply: Thanks for your suggestion. We have expanded the section and moved it before the "design experience" section.

*(3) Section I: In formal verification for HLS, how do you separate functional vs. timing bugs. This is something that seems to be overlooked.*

Reply: We added a new section (Section VII) to discuss simulation and formal verification. Simulation-based method can catch and debug both functional and timing errors (esp. for I/O timing). We are not aware of formal equivalence checking tools (or research publications) that specifically address timing bugs for HLS. We would be glad to include them if the reviewer can provide further suggestion.

*(4) Section II: When discussing the evolution of HLS, it would seem appropriate to classify or categorize the systems so that you are only presenting examples of such systems and techniques. This will help to differentiate the systems and later demonstrate why the new HLS will succeed.*
Reply: Thanks a lot for your comment. Following your suggestion, we have rewritten Section II. The tools are categorized according to their key contributions and timelines, and discussions on each specific category is added to provide more insight and to the trend.

*(5) In Section B, are the C-based languages restricted, how and why? Has that changed over time? I understand the desire to have algorithm designers to have a comfortable transition to FPGAs, but what about systems like BlueSpec?*
Reply: We have added more discussion about the pros and cons of C-based languages, as well as typical ways to expand/restrict the language to make it more suitable for HLS.
We have added BlueSpec reference in Section II. C-based HLS flow is currently more prevalent in the FPGA design community, and is the focus of this article.

*(6) Section III: AutoPilot (AP) creates synthesis reports. How good are the estimates for resource utilization, timing, latency, etc.? I ask because these estimates may be used to drive HLS design decisions before going through the design flow.*
Reply: Thanks for the question.
AutoPilot reports estimated performance (latency and clock period) and resource usage (e.g., LUT#, FF#, DSP#, BRAM#, etc.) as well as the utilization ratio of the target FPGA device. Although the reported numbers may not be accurate enough for design signoff yet, they do provide good basis for user to quickly explore the design space and make high-level design decisions.

*(7) In subsection A, you mentioned improved design quality. Mapping designs to ASICs is very different than mapping designs to FPGAs for structures like CAMs, gang-cleared bits, etc. If AP is an ASIC tool, how was it changed for FPGAs? If AP is an FPGA tool, is there a path to go to ASICs which will use different structures?*

*Section V: In the third paragraph, you mention the cost of implementing HW on FPGAs is different, but the implementation can also be different compared to ASICs. This ties into the earlier comment on how AP is customized for FPGAs vs. ASICs.*
*More specifically, how is AP customized for the Xilinx environment and individual FPGA platforms? There is quite a difference in architectures, moving from Virtex-4 to Virtex-5, for instance. DSP block functionality has also changed. Is the intermediate representation robust enough to handle Xilinx generational differences, mapping to Altera and/or ASICs?*
*This section also does not quantify how AP was changed to deal with Xilinx. It seems to me that significant modifications are required to integrate the tools and target Xilinx parts. This section also seems like a good place to discuss debugging. It is hardware to debugging highly-optimized C-code because the mappings from the assembly to C are difficult. This seems to be an even more significant issue for FPGAs.*
Reply: AutoPilot is able to support both ASIC and FPGA, by incorporating a flexible platform model. For components that are different in ASIC and FPGA, the platform library includes characterization of the components, and an automated mechanism to generate technology dependent cores is available.
In AutoPilot, each platform is characterized. Separate platform library files are included for Virtex-4 and Virtex-5. Differences like DSP functionality are also described in the platform library file and is considered in the synthesis process. AutoPilot also has characterization for Altera FPGA and some ASIC process libraries. Large and more complex built-in blocks on FPGA (like an integrated Ethernet controller) can be modeled as user IPs, and the mapping to user IP is controlled by the designer.
The approach for platform modeling in AutoPilot is described in Section V.A in the manuscript.

*(8) Section VI: Figure 2: re three intermediate results that can be used or studied? This way the user can provide input on the constraints as well, if they are not converging.*
Reply:
Soft constraints come from two sources: user directives (manual) and synthesis engine (automatic). Yes, soft constraints allow the user to enforce some degree of control on the synthesis result. We have expanded the subsection on scheduling with soft constraints.

*(9) There is no reference to Figure 3: There is a lot of text in the figure that is too small. Remove that text or make it smaller. I am not sure the figure is required.*
Reply: Thanks for your suggestion. We have redrawn the figure as well as several others to make them more visible.

*(10) Section VI: Subsections C and D are poorly written. There is very bad flow and no introduction for these subsections. Why are these subsections necessary? In introduction would help. In subsection D, Figure 4 is very poorly referenced. It would be good to provide more description of the various parts of the figure, only b and d are referenced. Please describe a, c, and e. There is also a very clear place to add a reference to 4.f.*
Reply: Subsections C and D are intended to describe some recent algorithmic developments on HLS that take special consideration of the FPGA platform. In the revised manuscript, we have added a brief description of the background and motivation in the beginning of each subsection. Figure 4 has been redrawn, with unreferenced subfigures removed.

*(11) In subsection E, how does AP know about the FPGA components? Does it read a technology file for the FPGA that it is going to map the design to?*
Reply: Yes. As described in the reply of (7), AutoPilot has detailed platform characterization for each and every FPGA platform device family. The performance, area and power characteristics of components like DSP and memory are described in the technology library file.

*(12) Section VII: Figure 5: Could you make the text more readable or remove it. The same applies to Figure 7.*
Reply: Thanks for the suggestion. We have redrawn the figures to make them more visible.

*(13) Section VIII: What modifications were made to make an efficient AP implementation? What were the modifications? Is this the addition of the pragmas? You have two different types of pragmas, for memories and to force mappings? When are they necessary and not? Is this complete?*
Reply: Code modifications typically include two steps: (a) change the code to make it synthesizable (including the elimination of dynamic memory allocation, avoid recursive function calls, etc.), (b) add pragmas/directives to guide the synthesis.
Memory selection and functional unit mapping are among the important pragmas we use in the designs presented. These pragmas are used to allow designer to guide the synthesis tool. Pragmas are not always necessary, because there are automatic mechanisms in the synthesis engine to make various decisions; and pragmas/directives are just one way to expose the decision engine to the designer.
Considering that there are different types of decisions to make in the synthesis process, there are also many types of pragmas. The complete list of pragmas/directives is long, and is not the focus of this paper. The AutoPilot User's Manual contains more details on this topic.

*(14) There is no reference to Figure 8. This would be helpful to describe the figure and make the font larger and/or clearer.*
Reply: Thanks for pointing out. We have redrawn the figure and added descriptions for it.

*(15) Are the results in Table 4 significant? There is less than a 10% difference. This could be attributed to a lot of ISE parameters or other things. A better comparison should be provided or explanation of why 5.6% vs. 5.9% is significant.*
Reply: Thanks for your comment. We have added additional clarification in Section IX. The DQPSK RTL design was created by experienced FPGA designers. It made use of highly optimized Xilinx CoreGen IPs when applicable (e.g. Viterbi decoder). In this particular case, producing comparable or slightly better area from C code via automated synthesis is not an insignificant result.

*(16) Section IX: Design validation and debugging should be moved to the beginning of the paper. IMHO, HLS will not gain traction until this is addressed. You provided some suggestions, but that has not changed in the lifetime of HLS.*
Reply: Thanks for your comment. We have added a section in the revised manuscript to describe design validation and debugging. Simulation-based approaches and formal approaches are both included. This is clearly a direction prevalent HLS tools are going.