Programming Abstractions in the Reconfigurable Hardware Era

Paulo Garcia, Heriot-Watt University
Robert Stewart, Heriot-Watt University

Programming abstractions decrease the cognitive gap between program idealization and expression. This high-level expressive power is achieved through layered abstractions - virtual machines, compilers, operating systems - which translate, at design and runtime, programmer-visible code into hardware-compatible code. While this paradigm is ideal for static, i.e., unmodifiable hardware, several problems arise when programming configurable (design time) and reconfigurable (runtime) hardware, where several abstractions break down. State of the art hardware/software co-design techniques (e.g., High Level Synthesis, Intermediate Fabrics) are, for the most part, *ad hoc* patches to the traditional abstraction stack, applicable only to specific toolchains or software components.

In this paper, we survey current programming and hardware/software co-design abstractions. We perform a systematic analysis of how the sub-systems responsible for each abstraction are affected by hardware reconfiguration and how novel design and runtime stacks must be architected in order to thrive in the reconfigurable hardware era. We show that the paradigm in inexorably moving towards a stage where every aspect of computing - languages, compilers, interpreters, operating systems and middleware - will encompass reconfigurable hardware.

CCS Concepts: •**Hardware** → **Reconfigurable logic and FPGAs; High-level and register-transfer level synthesis; Hardware description languages and compilation; Logic synthesis;** •**Software and its engineering** → **General programming languages; Architecture description languages;**

ACM Reference Format:
Paulo Garcia, Robert Stewart, 2016. Programming Abstractions in the Reconfigurable Hardware Era *ACM Comput. Surv.* 9, 4, Article 39 (March 2016), 14 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Abstractions are one of the fundamental aspects of computing. The advent of high level languages [] raised the level of abstraction from assembly programming and brought two main advantages to programmers: portability and productivity. No longer bound by processor-specific assembly instructions, code could be easily ported across different platforms, as long as a compiler were available. Processor agnosticism meant programmers could think in terms of the problem, rather than concerning themselves with architecture-specific details ("does this instruction affect the condition codes?"). It wasn't long until compilers could produce more efficient code than human programmers [], by optimizing at a higher level than assembly code: another advantage of abstractions.

Consider a contemporary high level language, e.g., Java []. A Java application can be pre-compiled on a completely different machine than the one on which it will run. The Java Virtual Machine (JVM) abstracts the underlying systems from the application programmer: both hardware and software. The JVM programmer, however, enjoys fewer abstractions: the JVM must be compiled targetting a specific Operating System

This work is supported by

(OS) and processor (e.g., IA32 []). However, it is still abstracted from many hardware details: particular micro-architectures (e.g., Intel i5 or i7 []) are invisible to the JVM, since these are abstracted by the OS. These abstractions are tried and tested on static, i.e., unmodifiable hardware: Application Specific Integrated Circuits (ASIC). Notice that this software stack is common even in resource constrained embedded systems, most of which run an embedded OS - typically a Linux kernel variant [] - and that *all* layers can be modified after system deployment; it is straightforward to write and compile code *in situ*, including OS kernel modules [].

Now consider the growing deployment of systems based on Field Programmable Gate Arrays (FPGAs) [], often used to implement complete System-on-Chip (SoC) solutions []. Parameterisable processors [], libraries of Intellectual Property (IP) blocks [] and custom components developed in Hardware Description Languages (HDL) [] enable system designers to fine-tune their platform for any given application []; at the cost of added software complexity. Modern FPGAs are capable of Dynamic Partial Reconfiguration (DPR) []: chip partitions can be reconfigured at runtime, in order to obtain power/performance benefits [], or merely to save on-chip space. This technology will likely improve, enabling more and more fine-grained reconfiguration capabilities [].

What happens when hardware reconfiguration reaches the same level of configurability and programmability that software currently enjoys? Certainly one can envision several advantages. The hypothetical Java application could reconfigure hardware to deploy an accelerator: a custom circuit that performs computations in shorter time or consuming less power. But this request must certainly go through the JVM and through the OS, since only at the OS layer are hardware details known: what the interfaces to communicate with the accelerator are. This breaks, to a certain degree, the abstraction model; to the OS, the underlying layer - hardware - which it's supposed to abstract, is modified by an upper layer - application -. This accelerator could be used to snoop and modify data in memory buses, giving the application read/write access to memory locations it shouldn't access: nullifying the spatial separation the OS is supposed to enforce [].

Throughout the remainder of this paper, we survey the state of the art in the different aspects of programming abstractions for reconfigurable hardware. We begin by revising abstractions and hierarchies in static hardware in Section 2. These are described from the *system* perspective: the purpose and abstractions offered by each software stack layer; and from the *design* perspective, namely programmer concerns and the associated programming language paradigms. We continue in Section 3 by reviewing hardware/software co-design for configurable (i.e., at design time) hardware, showing the current techniques, tools and languages for mixed design and their impact on software. In Section 4, we survey the state of the art in reconfigurable co-design, where we present current work on runtime systems and associated design paradigms for runtime reconfiguration and their implications. In Section 5, we describe what we believe is the inexorable future of completely reconfigurable systems, based on current research directions.

To the best of our knowledge, this is the first paper to describe a complete analysis of research and roadmaps in programming abstractions in reconfigurable hardware.

2. PROGRAMMING ABSTRACTIONS IN STATIC HARDWARE

The conventional view on *programming abstractions* -the level at which software is designed, and which concerns afflict the programmer- is a hierarchical one, depicted in Fig. **??**. At the highest level, *cloud computing*: software is designed at a level where the both the underlying Operating System (OS) and hardware platform are invisible; even the geographical location of the machines that run such code, or the number of machines concurrently running such code, is unknown. Software is encapsulated with and within services [] and Application Programming Interfaces (APIs) [] and managed by the underlying software stack (encompassing Virtual Machines, Hypervisors, OSs, Middlewares, etc.) at runtime. Examples of languages for such an abstraction level are PHP [] and JavaScript **??**.

One step down the hierarchy, there is interpreted (sometimes called managed []) code []. Software runs on top of interpreters or virtual machines, allowing complete portability across different OSs and architectures. Like cloud computing, managed code is not aware of the underlying OS and hardware platform; however, it is more self-contained (i.e., less reliant on APIs and services) than cloud software. Examples of languages for this layer are Java [] and C# [].

The next step is native application code. This code is compiled for a particular OS and for a particular architecture (e.g., ARM, IA64), but is unaware of particular micro-architectural details (e.g., how the memory system is organized). Examples of languages for this layer are Haskell [] and C/C++ [].

3. HARDWARE SOFTWARE CO-DESIGN

3.1. High Level Synthesis

High-Level Synthesis (HLS) promises to boost hardware design productivity into software-like levels, ending the long development and verification processes typically associated with Register Transfer Level (RTL) design. The momentum behind the HLS movement is primarily fueled by two aspects: on one hand, the hardware community desires to increase productivity and ensure design correctness in order to guarantee the time-to-market of complex SoCs keeps up with Moore's law. On the other hand, the software community, lacking RTL-savyy, increasingly adopts FPGAs in fields as diverse as high performance computing or image processing, and wishes to program them using established software languages/frameworks. The appearance of *platform FPGAs*, which are essentially SoCs combining general-purpose processor(s) and/or other "hard" blocks with "soft" FPGA fabric, increased the demand for powerful HLS even further, thanks to opportunities for hardware-software co-design.

Despite approximately three decades of HLS development, we are still far from the levels of productivity achieved in the software world. The HLS landscape, shaped by FPGA vendors' proprietary toolchains, third-party commercial toolchains and academic/open-source projects, is made up of more than forty different HLS frameworks, none of which has gained widespread adoption: Verilog and VHDL are still the *de facto* development languages for FPGAs. The goal of this survey is to examine the reasons behind this and point out future research directions for HLS.

HLS research has been previously summarized from different perspectives: [Martin and Smith 2009], [Nane et al. 2016], [Trimberger 2015], [Meeus et al. 2012] and [Cong et al. 2011] have described the historical evolution of HLS tools, primarily focusing on industry adoption. [Zhang and Ng 2000], [Compton and Hauck 2002] and [Cardoso et al. 2010] focus on the dynamic-reconfiguration support of HLS tools. We summarize the field from a different perspective, filling a gap in the literature; namely *HLS languages' abstractions*, focusing on the clash of hardware and software traditional views.

The diverse nature of HLS languages/toolchains (some are novel full-fledged languages such as Bluespec, others subsets of legacy languages such as C, and others are domain specific embedded languages, such as Cλash) complicates this analysis; we treat each as comparable languages *per se* when appropriate, and we make very fine distinctions in other cases. Different toolchains which use the same underlying language are also treated distinctly: throughout this manuscript, we use the terms *language* and *toolchain* interchangeably when referring to HLS.

Hardware Description Languages (HDL), such as Verilog and VHDL, lack many of the features typically associated with high level languages. There is no complex type system: every signal is a bit array (integer types are merely placeholders for a specific bit width). There are very few syntactical constructs: the various loop and type constructs typically found in software make no sense. This is because HDLs specify the behavior, not as a sequential computation (where parallelism must be explicitly managed through threads/processes) but as a computation for every single moment in time (behavior at each clock cycle) where parallelism is implicit. Loop unrolling is seen as a compiler optimization in C, while in HDL, it is built into the language semantics: a loop that cannot be fully unrolled at design time will result in a synthesis error. This paradigm shift from temporal flexibility to temporal strictness is one of the main reasons software programmers struggle with HDLs, and one HLS tools attempt to abstract.

However, this temporal paradigm is one of the greatest strengths of HDLs and one of the weaknesses of HLS (we will elaborate on this in the following sections); HDLs can cope with complex timing requirements, such as refresh rates for DRAM memories, communication protocols, or circuit initialization procedures. When HLS languages fail to provide mechanisms to perform these procedures efficiently and clean interfaces to HDL code, there is little motivation for hardware engineers to adopt such a toolchain. This advents from the fact that many HLS languages are datapath-oriented, rather than control-oriented (HDLs are both).

Clock signals are arguably the most important signals in an FPGA design, and thus, they are explicitly stated in HDLs. Designers can distribute complex designs across asynchronous clock domains, perform clock gating for power reduction and keep an accurate record of elapsed time (every hardware designer has had to implement a clock cycles counter for some low-level communication protocol at one time). Most HLS languages, however, do not have an explicit clock signal. Either each language construct operates in one clock cycle or, as is the case in some dataflow languages, there is no programmer-visible concept of time. To the best of our knowledge, very few HLS toolchains are yet capable of handling multiple clock domains automatically, inserting appropriate synchronization logic (e.g., [Lhairech-Lebreton et al. 2010]).

3.1.1. Imperative Languages. C is the most familiar software language for hardware designers. Hence, it is not surprising that many HLS toolchains use C, and other similar imperative languages and Object-Oriented (OO) variants, as a foundation (although many software programmers would disagree C should be considered ”high level”). Table I depicts imperative-based HLS toolchains. It is by no means complete, but suffices to grasp the ammount of effort invested in imperative languages to FPGA design.

There are three main advantages to this flavour of HLS: *familiarity, control* and *co-design*. The familiar syntax allows programmers to express algorithms quickly, often re-using code. Tried and tested software applications can be migrated to hardware for acceleration. The sequential semantics of C-like languages, rich in loop and conditional constructs, lends itself well to the implementation of protocols and control operations. Since software and hardware are described in the same language, design space exploration strategies can be employed to meet design performance, power and area constraints, mapping an algorithm to CPU(s)/FPGA hybrid systems. This paradigm is especially favorable when targetting platform FPGAs; hence the extensive support offered by FPGA vendors (e.g., Xilinx vivado HLS and SDK).

This paradigm is not without several drawbacks. It is notoriously difficult to explicitly express parallelism in imperative languages; it must typically be expressed

Table I. Imperative/OO HLS

Language	Source
SystemC	[Panda 2001]
Handel-C	[Loo et al. 2002]
OCAPI-XL	[Vanmeerbeeck et al. 2001]
Catapult-C	[Bollaert 2008]
Vivado HLS	[Feist 2012]
Impulse C	[Xu et al. 2010]
C-to-Silicon	[Cadence 2011]
Synphony C	[Synopsis 2011]
Cynthesizer	[Cadence 2014]
LegUp	[Canis et al. 2011]
ASC	[Mencer 2006]
Altera C2H	[Nios 2007]
CHiMPS	[Putnam et al. 2008]
ROCCC	[Villarreal et al. 2010]
GAUT	[Coussy et al. 2010]
Trident	[Tripp et al. 2007]
Altera SDK for OpenCL	[Settle 2013]
Xilinx SDAccel	[Fifield et al. 2016]
FCUDA	[Papakonstantinou et al. 2009]
LIME	[Auerbach et al. 2010]
KIWI	[Singh and Greaves 2008]
DWARV	[Nane et al. 2012]
Bambu	[Pilato and Ferrandi 2013]
Hercules	[Kavvadias and Masselos 2015]

Table II. Functional HLS

Language	
Clash	[Harmsen 2012]
HML	[Li and Leeser]
ForSyDe	[Sander et al. 2009]
Lava	[Singh and Sheeran 2004]
PARO	[Hannig et al. 2008]
Esterel	[Hammarberg and Nadjm-Tehrani 2003]
MMAlpha	[Derrien and Risset 2000]
Verity	[Aguilar-Pelaez et al. 2014]
ReWire	[Procter et al. 2015]
SAFL	[Sharp 2004]
Hume	[Sérot and Michaelson 2012]

through compiler directives (*pragmas*) for loop unrolling, which rely on compiler optimizations. Granularity is an issue: most HLS compilations operate at the function granularity, which might force legacy software to be re-written in order to encapsulate hardware-destined and software-destined code separately. Timing is an issue, as it may be impossible to predict how many clock cycles a particular language construct will take to execute, depending on the HDL generation strategy. As these languages were built for Von Neumann machines (implying a large, shared memory space), many language features are not directly amenable to hardware synthesis (pointers and dynamic memory allocation are notorious examples).

3.1.2. Functional Languages. Advocates for functional programming have developed several HLS functional flavours, either through new complete languages or through small languages embedded in consolidated ones such as Haskell. Table II depicts an overview of functional languages targetting FPGAs.
♣**PG:** advantages and problems♣

3.1.3. Domain Specific Languages. In an effort to avoid the challenges of synthesizing complete complex languages, several Domain Specific Languages (DSL) have emerged in recent years. DSLs are well known in the software domain: they allow programmers to express solutions at a higher abstraction level than typical Turing-complete languages, within the semantics of the particular application domain.

In the context of HLS, DSLs have several other advantages. The limited syntactical constructs are more easily synthesizable; in other words, it is far simpler to ensure complete language coverage. Knowledge of the application domain allows the use of hardware templates optimized for the domain: rather than generic hardware structures designed for flexibility, the HLS tool is free to infer specialized architectures, optimized for power, performance and area within the domain (e.g., in synchronous dataflow DSL synthesis, the HLS compiler is aware of the timing relationships between modules and can infer pipeline stages separated by registers; in asynchronous dataflow, the HLS compiler is forced to infer FIFOs between modules).

Darkroom [Hegarty et al. 2014] is language and compiler for image processing. Its semantics allow it to synthesize line-buffered pipelines, with all intermediate values in local line-buffer storage. Images at each stage of computation are specified as pure functions from 2D coordinates to the values at those coordinates, declared using a lambda-like syntax. In the first version, it only supports programs that are straight pipelines with one input, one output, and a single consumer of each intermediate. HIPA^{cc} [Membarth et al. 2016] (Heterogeneous Image Processing Acceleratio) is another DSL for image processing. It is a C++ embedded DSL [Cuadrado and Molina 2007], which uses the LLVM backend [Lattner and Adve 2004] for software code generation, and C code annotated with pragmas for Vivado HLS.

RVC-CAL [Wipliez et al. 2011] is an asynchronous dataflow language which possesses backends for FPGA, e.g., Xronos [Bezati et al. 2013]. RVC-CAL is based on dataflow process networks with the addition of firing rules. Xronos uses Orcc compiler [Yviquel et al. 2013] as its front-end, which parses RVC-CAL actors and generates and intermediate representation suited to its OpenForge backend. Another streaming DSL is Optimus [Hormati et al. 2008], based on the StreamIt language [Thies et al. 2002]. A common feature of both languages is that embedded memories are used to implement local arrays and other data structures used by the filters. Thus, for large stream graphs, embedded memories quickly become the bottleneck resource.

Spiral [Püschel et al. 2005] is a code generation system for Digital signal Processing (DSP) transforms which has been extended to generate DSP IP cores for FPGA [D'Alberto et al. 2007].

3.2. New Generation Hardware Description Languages

Three new generation HDLs are particularly relevant: Bluespec [Nikhil 2004], Chisel [Bachrach et al. 2012] and Cx [Synflow 2014].

Bluespec extends system Verilog to provide a higher level of abstraction. Interfaces are a core construct of Bluespec. Interfaces group signals according to methods, which define the semantics of access to a signal and can be used to parameterize modules upon instantiation. Rather than ”always” or ”process” constructs familiar to Verilog/VHDL designers, Bluespec defines behavior through rules (i.e., guarded actions) which specify how data are moved from state to state.

Chisel is an HDL embedded in the Scala language which supports multiple design paradigms, including object orientation, functional programming, parameterized types, and type inference. Two notable features are the capability to specify composite types (i.e., C-like *structs*) and automatic inference of bit-widths, unlike strict bit-width definitions in legacy HDLs. OO-like inheritance allows modules to be re-used in the definition of higher-order modules without the messy sub-module instantiation (and corresponding ”rats nest” wiring). Computations can be expressed in functionalfriendly constructs such as *map* and *fold* and the expressive generator systems simplifies the re-use of parameterizable modules.

Cx offers a highly structured syntax with strong bit-accurate static typing. A Cx design is described as a set of sequential tasks connected together and executed concurrently, where dependency injection and inheritance can be applied to tasks. The Cx compiler generates human readable Verilog/VHDL code or C code for verification. Cx systems are described as Kahn Process Networks [Edwards 2000] where connections are inferred by discrete non-interruptible execution rules. Unlike legacy HDLs, clocks are not explicitly declared within code, but language semantics strictly specify the timing behavior of language constructs, providing cycle aware design much like VHDL and Verilog.

Bluespec, Chisel and Cx are greatly superior to legacy HDLs. Software concepts such as inheritance and type composition have found the way to HDLs, reducing the semantic gap between concept and implementation. Rather than leveraging existing software languages for FPGA synthesis (with the associated semantic problems), new generations HDL incorporated high-level language concepts in languages fine-tuned for hardware design. However, these new generation HDLs are still far from producing the type of disruptive innovations expected from HLs, primarily due to three reasons:

— They follow the same design principle as decades-old legacy HDLs: FPGAs as a self contained entity. This contrasts the approach offered by FPGA vendors, who provide software suites that allow design at board, rather than chip, level. These suites are made up of several stacks which, through a complex design process involving constraint files, IP libraries, proprietary buses, etc, generate FPGA designs incorporating peripheral devices access and software interaction. It would be expected of new generation HDLs to incorporate more complex constructs, modeling off-chip interfaces within the semantics.
— They do not incorporate the Von Neumann notion of memory. A substantial portion of state of the art FPGA systems incorporate external memory to accommodate data requirements. On new generation HDLs, this must be handled as in legacy HDLs: the programmer is responsible for interfacing with memory and managing data transmission to and from, burdening them with implementation details independent of the top level computation. It would be expected of new generation HDLs to model memory transparently (e.g., by specifying memory-allocated data as a built-in type) and generate hardware to transfer to and from memory seamlessly, through an on-chip hierarchy (physical constraints such as which FPGA pins are connected to memory can be resolved at linking stage, prior to synthesis).
— There are no semantic considerations for software interface. With the rise of the platform FPGA, it would be expected that new generation HDLs would borrow concepts from research in hardware-software interface research (e.g., PushPush [Fleming et al. 2015]) in order to provide semantic mechanisms for incorporating typed functions for bi-directional interaction with software objects.

3.3. Expressiveness

HLS languages semantics limit which circuit functionalities can be expressed efficiently, or at all. A notorious example is the inference of tri-state buffers. Many HLS languages generate tri-state buffers within their HDL output for connecting functional elements; i.e., they are explictly stated in the HLS compiler HDL template. E.g., a HLS toolchain can instantiate functions as hardware accelerators, accessed through a common bus (LegUp [Canis et al. 2011]) uses the Avalon bus [Altera 2002] which implements a point to point, rather than shared, interconnect, implemented with multiplexers rather than tri-state buffers). Infering tri-state buses from user logic, however, is non-trivial. Implementations where several modules must communicate in a master-slave fashion, which hardware designers would naturally implement through shared buses implemented through tri-state logic, will result in several point to point connections in a naive HLS implementation.

In imperative HLS, inputs and outputs are derived from function arguments (inputs) and function return value (outputs); both are intrinsically uni-directional. Infering tri-state buffers would require compiler directives on function declaration, extensive knowledge of the underlying HDL code generation to identify bus sharing opportunities (breaking the abstraction layer offered by HLS) and compiler optimizations to infer timing semantics for shared bus access. HLS semantics could perhaps specify shared memory, accessed through pointers and controlled through familiar software constructs such as mutexes and semaphores, to infer tri-state based buses and generate more area-efficient systems.

In functional HLS,

♣**RS:** Rob?♣

Tri-state buffers are a good example of a hardware feature that cannot be easily expressed by HLS language semantics. The opposite aspect is perhaps more researched: language semantics that cannot be (easily) synthesized to hardware. In imperative languages, all non-synthesizable constructs boil down to the same reason: memory. Imperative languages assume the Von Neumann model, which is not true on FPGAs. When functions are compiled to hardware modules, with arguments and return values specifying inputs/outputs, all local data are instantiated as wires, registers or embedded memory blocks: all encapsulated in the module. Global variables break this paradigm; they can only be resolved by instantiating logic/storage at a higher hierarchical level and adding connections between all modules that use them (meaning that modules' inputs/outputs no longer match function header). Further challenges arise in routing, if this data are shared by several modules, possibly decreasing clock frequencies, and in scheduling: while in (single-thread) software there is a strict sequential ordering of operations, this is not true in FPGA implementations. Either the HLS compiler extracts timing behavior from code analysis and places appropriate mechanisms on the hardware scheduler to preserve behavior, or it must force the use of mutexes/semaphores to control access (and generate equivalent hardware). The problems with pointers derive from the same aspects, whether they point to global variables or to local variables within other functions.

Recursion is again hindered by the lack of memory, namely the implicit stack. Frame pointers and stack pointers used by the C runtime to manage function activation records make no sense in an FPGA implementation without a program counter. A function cannot call another instance of itself, since data are placed in physical locations, not in an addressable memory. Runtime polimorphism (even if implemented through a low-level mechanism such as C function pointers) cannot be accomplished when modules are connected point to point: it would require a bus or Network-on-Chip (NoC) scheme, where each module can be addressed (i.e., implementing the hardware equivalent of a typed procedure call).

In functional HLS,

♣**RS:** Rob: recursive functions are a hot topic, and monads as well♣

3.3.1. Hierarchy and Modularity. Hierarchy is one of the foundations of hardware design. Hardware designers develope small modules which are re-used, integrated and composed (with no small ammount of "glue logic") to create complex circuits. There are three main advantages in hierarchy: *ease of verification*, *algorithm decomposition* and *implicit parallelism*. Small modules can be debugged and verified easily; when integrated into a higher-order module, functionality can be trusted, simplifying the verification of added logic. The algorithm to be implemented can be decomposed into several more atomic components, each implemented individually, minimizing the cognitive effort. Composing a higher-order module from several lower-level ones, each operating in parallel, enforces highly efficient designs not constrained by sequential operations, when possible. Legacy HDLs offer several constructs supporting hierarchy, such as parameterizable sub-module instantiation.

However, hierarchy involves its own set of challenges. Managing code becomes more difficult: deeply nested modules are often connected to ones further up the hierarchy. This means that wires traverse several hierarchical levels, often across multiple source files. Any modification (e.g., adding a new signal) must be applied to all levels (typically referred to as ”rats nest” wiring problem). There is no semantic information apart from directionality (input or output) and bit width: particularly, there is absolutely no information about timing validity. New generation HDLs tackle this problem: in Bluespec, data signals are accompanied by ”ready” and ”enable” lines, adding semantic information to module interfaces.

In the HLS domain, language semantics dictate how hierarchy is established. In imperative languages such as C, hardware synthesis is typically performed at function granularity. Functions that call other functions are result in modules which are hierarchically composed by the modules generated from the called functions, thus maintaining the ease of verification and algorithm decomposition capabilities offered by HDLs. However, parallelism is not implicit, since language semantics are not parallel. Native semantics dictate that one sub-function will operate, pass the result to the top-level function, and the second sub-function will then operate, even if computations are independent. Parallelism can be achieved, either through compiler transformations, performing dependency analysis and scheduling optimizations, or through explicit compiler directives which guide the parallelization process. Neither is simple: either the synthesis process is complicated by extensive analysis, or more responsabilities are placed on the programmer, which must refactor their code to incorporate toolchain-specific pragmas (hindering the familiarity aspect of HLS and breaking standard language semantics).

Functional HLS languages do a much better job at composing circuits, since hierarchy is implicit in language semantics and there is no strict sequential behavior. Again, each function is mapped to a module. Function composition and higher-order functions (i.e., which take functions as arguments) creates a clean hierarchy of modules, where the functional paradigm lends itself very well to analyse data dependencies and extract parallelism; familiar concepts to functional programmers, who advocate the use of *map* and *fold* operators rather than loop constructs.

3.3.2. HDL Interoperability. HLS toolchains must allow seamless integration with HDL languages, both for legacy code re-use and for flexibility; some design aspects are complicated by HLS semantics and more easily resolved in HDL. [Whitham 2013] excelently describes the problems of integrating Bluespec with Verilog code. Notably, connecting legacy IP modules often require ”wrappers” which trasnlate the native module interface into Bluespec compliant semantics, complicating re-use. Some assumptions (e.g., active high or active low signals) cannot be changed from the HDL domain and require proper modification.

More generally, HLS languages must provide the kind of support to Verilog/VHDL that C compilers provide to assembly language. Hierarchical use of HDL modules within HLS semantics, similarly to inline assembly in C code (hierarchy), and seamless interconnection following flexible, configurable ABIs for linking (composition), which allow specifying interfaces, polarities and timing behavior in a flexible way. New generation HLS languages must treat interfaces, not just as port definitions such as in

HDLs, but as complex, flexible protocols with parameterizable ABIs to allows seamless interoperability.

3.3.3. I/O Interfaces. In the software world, the majority of programmers need not concern themselves with the target architecture (exceptions are embedded programmers who must interact directly with hardware). Software is developed in the same way for ARM, IA64 or PowerPC: the compiler toolchain handles appropriate assembly code generation and linking. This is not true for FPGAs. The equivalent of assembly code generation, generating configuration bistream files, is handled by the synthesis tool. But linking, which for software connects applications with initialization/bootstrap code, static and dynamic libraries, is manually performed by hardware designers in function of specific FPGAs (mapping system input/output ports to FPGA pins) and off-chip connections. This part of the design process limits the usage of many HLS toolchains by programmers lacking FPGA expertise, since low-level constraints requiring knowledge of place-and-route tools must be manually specified. HLS toolchains are limited to the FPGA boundaries.

With the rise of the platform FPGA, more and more designs will incorporate software running on off-chip processors and accesses to external memories and other peripheral devices. One of the greatest HLS breakthroughs will be automated porting across different FPGA families and boards. FPGA designs will be synthesized like the Linux kernel: specifying an architecture and board through configuration options prior to compilation. Linkers will automatically inject interconnection logic in the design, map system ports to FPGA pins, generate constraint files and invoke place-and-route tools.

It is still a utopic view, far from the current capabilities of HLS toolchains. But we are inexorably moving towards that goal. Perhaps by utilizing *intermediate fabrics* for runtime reconfiguration [Coole and Stitt 2010] or by introducing semantic information to HDL code which allows automatic I/O mapping through typed interfaces [Fleming et al. 2015]. Until then, programmers will be limited to vendor-specific ”plug-and-play” platforms or require FPGA-fluent engineers to manage the low level details of generated HLS code.

3.4. Hardware-Software Interactions

In light of the popularity of platform FPGAs, hardware-software interactions deserve particular attention. The most basic approach is to manually ”patch” software, replacing function calls with instructions for hardware access (which can be either dedicated instructions, added to a soft processor instruction set, or memory-mapped accesses, depending on how bespoke hardware is connected). More recently, FPGA vendors have partially automated software-hardware interactions (e.g., Xilinx toolchain allows adding AXI-compliant IP blocks to a design and facilitate generating software for interaction, even device drivers for Linux). Projects such as LegUp [Canis et al. 2011], which (semi)automatically profile a software application running on a soft MIPS processor and generate hardware, at function granularity from C source code, connect hardware accelerators through a dedicated Avalon bus and patch software to access them. Both approaches, followed by many other HLS toolchains, treat software as a first class citizen and hardware as second class; i.e., software controls the execution flow and can utilize hardware accelerators whenever it sees fit, an approach commonly referref to as *vertical integration*. *Data transport layers* enhance the semantics of interoperability by providing bi-directional channels between hardware and software: examples are Xillybus [Xillybus 2016], LEAP [Adler et al. 2011] and RIFFA [Jacobsen et al. 2015]. Although more powerful and flexible than vertical integration approaches, these technologies still require explicit function linking by the programmer. A more sophisticated technology is PushPush, which exposes component functionality (where a component can be hardware or software) as strongly typed functions, and allows any component to access functions exposed by any other in the system [Fleming et al. 2015], allowing hardware blocks to invoke software functionalities seamlessly; linking is performed automatically, either at system integration or runtime, implementing *horizontal integration*.

4. RECONFIGURABLE RUNTIME SYSTEMS

5. THE NEXT ERA: FULLY RECONFIGURABLE SYSTEMS

6. CONCLUSIONS AND FUTURE DIRECTIONS

ACKNOWLEDGMENTS

We acknowledge the support of the Engineering and Physical Research Council, grant references EP/K009931/1 (Programmable embedded platforms for remote and compute intensive image processing applications) and EP/J015180/1 (Sensor Signal Processing).

REFERENCES

Michael Adler, Kermin E Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 25–28.

Eduardo Aguilar-Pelaez, Samuel Bayliss, Alex Smith, Felix Winterstein, Dan R Ghica, David Thomas, and George A Constantinides. 2014. Compiling higher order functional programs to composable digital hardware. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 234–234.

Altera. 2002. Avalon Bus Specification. (2002). http://coen.boisestate.edu/clarenceplanting/files/2011/09/avalon_bus_spec.pdf

Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *ACM Sigplan Notices*, Vol. 45. ACM, 89–108.

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1216–1225.

Endri Bezati, Marco Mattavelli, and Jörn W Janneck. 2013. High-level synthesis of dataflow programs for signal processing systems. In *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*. IEEE, 750–754.

Thomas Bollaert. 2008. Catapult synthesis: a practical introduction to interactive C synthesis. In *High-Level Synthesis*. Springer, 29–52.

Cadence. 2011. C-to-Silicon Compiler High-Level Synthesis. (2011). https://www.cadence.com/rl/Resources/datasheets/C2Silicon_ds.pdf

Cadence. 2014. Cynthesizer Solution. (2014). http://www.cadence.com/rl/Resources/datasheets/cynthesizer_ds.pdf

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 33–36.

João MP Cardoso, Pedro C Diniz, and Markus Weinhardt. 2010. Compiling for reconfigurable computing: A survey. *ACM Computing Surveys (CSUR)* 42, 4 (2010), 13.

Katherine Compton and Scott Hauck. 2002. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)* 34, 2 (2002), 171–210.

J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (April 2011), 473–491. DOI: http://dx.doi.org/10.1109/TCAD.2011.2110592

James Coole and Greg Stitt. 2010. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*. IEEE, 13–22.

Philippe Coussy, Ghizlane Lhairech-Lebreton, Dominique Heller, and Eric Martin. 2010. GAUT–a free and open source high-level synthesis tool. (2010).

Jesús Sánchez Cuadrado and Jesús García Molina. 2007. Building domain-specific languages for model-driven development. *Software, IEEE* 24, 5 (2007), 48–55.

Paolo D'Alberto, Peter A Milder, Aliaksei Sandryhaila, Franz Franchetti, James C Hoe, José MF Moura, Markus Puschel, and Jeremy R Johnson. 2007. Generating fpga-accelerated dft libraries. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*. IEEE, 173–184.

Steven Derrien and Tanguy Risset. 2000. Interfacing compiled FPGA programs: the MMAlpha approach. In *PDPTA*.

Stephen A Edwards. 2000. Kahn process networks. In *Languages for Digital Embedded Systems*. Springer, 189–195.

Tom Feist. 2012. Vivado design suite. *White Paper* 5 (2012).

Jeff Fifield, Ronan Keryell, Hervé Ratigner, Henry Styles, and Jim Wu. 2016. Optimizing OpenCL applications on Xilinx FPGA. In *Proceedings of the 4th International Workshop on OpenCL*. ACM, 5.

S. T. Fleming, I. Beretta, D. B. Thomas, G. A. Constantinides, and D. R. Ghica. 2015. PushPush: Seamless integration of hardware and software objects via function calls over AXI. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. DOI:http://dx.doi.org/10.1109/FPL.2015.7294024

Jerker Hammarberg and Simin Nadjm-Tehrani. 2003. Development of safety-critical reconfigurable hardware with Esterel. *Electronic Notes in Theoretical Computer Science* 80 (2003), 219–234.

Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. 2008. PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 287–293.

Ruud Harmsen. 2012. Compiling Recursion to Reconfigurable Hardware using CLaSH. (2012).

James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. (2014).

Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. 2008. Optimus: efficient realization of streaming applications on FPGAs. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 41–50.

Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. 2015. RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 8, 4 (2015), 22.

Nikolaos Kavvadias and Kostas Masselos. 2015. Source and IR-level optimisations in the HercuLeS high-level synthesis tool. *International Journal of Innovation and Regional Development* 6, 3 (2015), 243–266.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86.

G. Lhairech-Lebreton, P. Coussy, and E. Martin. 2010. Hierarchical and Multiple-Clock Domain High-Level Synthesis for Low-Power Design on FPGA. In *2010 International Conference on Field Programmable Logic and Applications*. 464–468. DOI:http://dx.doi.org/10.1109/FPL.2010.94

Yanbing Li and Miriam Leeser. HML: an innovative hardware description language and its translation to VHDL. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC'95/CHDL'95/VLSI'95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal*. IEEE, 691–696.

SM Loo, B Earl Wells, N Freije, and J Kulick. 2002. Handel-C for rapid prototyping of VLSI coprocessors for real time systems. In *System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium on*. IEEE, 6–10.

Grant Martin and Gary Smith. 2009. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers* 4 (2009), 18–25.

Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. 2012. An overview of todays high-level synthesis tools. *Design Automation for Embedded Systems* 16, 3 (2012), 31–51.

R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Krner, and W. Eckert. 2016. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (Jan 2016), 210–224. DOI:http://dx.doi.org/10.1109/TPDS.2015.2394802

Oskar Mencer. 2006. ASC: a stream compiler for computing with FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 25, 9 (2006), 1603–1617.

Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. 2012. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 619–622.

R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP, 99 (2016), 1–1. DOI:http://dx.doi.org/10.1109/TCAD.2015.2513673

Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 69–70.

II Nios. 2007. C2H Compiler Users Guide. *Altera, May* (2007).

Preeti Ranjan Panda. 2001. SystemC-a modeling platform supporting multiple design abstractions. In *System Synthesis, 2001. Proceedings. The 14th International Symposium on*. IEEE, 75–80.

Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*. IEEE, 35–42.

Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 1–4.

Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. 2015. Semantics Driven Hardware Design, Implementation, and Verification with ReWire. *SIGPLAN Not.* 50, 5, Article 13 (June 2015), 10 pages. DOI:http://dx.doi.org/10.1145/2808704.2754970

Markus Püschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gačic, Yevgen Voronenko, and others. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

Andrew Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, Prasanna Sundararajan, and Susan Eggers. 2008. CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 173–178.

Ingo Sander, Alfonso Acosta, and Axel Jantsch. 2009. Hardware design and synthesis in ForSyDe. In *Workshop on Hardware Design using Functional languages (HFL 09)*.

Jocelyn Sérot and Greg Michaelson. 2012. Harnessing parallelism in FPGAs using the hume language. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 27–36.

Sean O Settle. 2013. High-performance dynamic programming on FPGAs with OpenCL. In *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*. 1–6.

Richard Sharp. 2004. 5. High-Level Synthesis of SAFL. In *Higher-Level Hardware Synthesis*. Springer, 65–86.

Satnam Singh and David Greaves. 2008. Kiwi: Synthesis of FPGA circuits from parallel programs. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 3–12.

Satnam Singh and Mary Sheeran. 2004. Designing FPGA circuits in Lava. *Unpublished paper, http://www. gla. ac. uk/˜ satnam/lava/lava_intro. pdf* (2004).

Synflow. 2014. Introducing Cx. (2014). http://cx-lang.org/

Synopsis. 2011. Synphony C Compiler. (2011). https://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx

William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, 179–196.

Stephen M Trimberger. 2015. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proc. IEEE* 103, 3 (2015), 318–331.

Justin L Tripp, Maya B Gokhale, and Kristopher D Peterson. 2007. Trident: From high-level language to hardware circuitry. *Computer* 3 (2007), 28–37.

Geert Vanmeerbeeck, Patrick Schaumont, Serge Vernalde, Marc Engels, and Ivo Bolsens. 2001. Hardware/software partitioning of embedded system in OCAPI-xl. In *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*. IEEE, 30–35.

Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. 2010. Designing modular hardware accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 127–134.

Jack Whitham. 2013. What's Bad about Bluespec System Verilog. (2013). http://blog.jwhitham.org/2013/07/whats-bad-about-bluespec-system-verilog\%_10.html

Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. 2011. Software code generation for the RVC-CAL language. *Journal of Signal Processing Systems* 63, 2 (2011), 203–213.

Xillybus. 2016. An FPGA IP core for easy DMA over PCIe with Windows and Linux. (2016). http://xillybus.com/downloads/xillybus_product_brief.pdf

Jimmy Xu, Nikhil Subramanian, Adam Alessio, and Scott Hauck. 2010. Impulse C vs. VHDL for accelerating tomographic reconstruction. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 171–174.

Hervé Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickaël Raulet. 2013. Orcc: Multimedia development made easy. In *Proceedings of the 21st ACM international conference on Multimedia*. ACM, 863–866.

Xuejie Zhang and Kam W Ng. 2000. A review of high-level synthesis for dynamically reconfigurable {FPGAs}. *Microprocessors and Microsystems* 24, 4 (2000), 199 – 211. DOI: http://dx.doi.org/10.1016/S0141-9331(00)00074-0

Received February 2016; revised March 2016; accepted June 2016