Accelerator Design with High-Level Synthesis

Christian Pilato and Stephanie Soldavini

Abstract Specialized accelerators can exploit spatial parallelism on both operations and data thanks to a dedicated microarchitecture with a better use of the hardware resources. Designers need to describe such components (including the resources, their interconnections, and the control logic) in proper hardware languages compatible with synthesis tools. This process requires hardware design skills that are uncommon in software programmers. To boost the use of spatial accelerators, software programmers need automated methods, like *high-level synthesis* (HLS), to specify hardware blocks with high-level languages and automatically translate their specifications into the corresponding hardware descriptions ready for synthesis. While HLS is a key enabling technology for the design of complex hardware/software architectures, developing efficient spatial accelerators requires efficient HLS methods to co-optimize performance and hardware cost with a hardware/software co-design approach. In this chapter, we present the current state of the art in high-level synthesis, covering all steps to create the specialized microarchitecture of an accelerator. We also discuss outstanding challenges that can be addressed with the use of HLS.

1 Introduction

Technology scaling is coming to an end due to physical limitations on building smaller transistors and keeping all of them active inside the chip (*dark silicon* problem) [1]. Designers need novel solutions to increase the performance and reduce the power consumption of computing systems [2]. Modern workloads, such as Big Data and machine learning applications, exacerbate these issues because they need

Christian Pilato and Stephanie Soldavini
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy; e-mail: christian.pilato@polimi.it, e-mail: stephanie.soldavini@polimi.it.
(Corresponding author: Christian Pilato)

to process an increasing amount of data to identify hidden relationships and extract valuable knowledge [3]. Traditional computing systems are not able to satisfy the performance requirements of these applications, while the large and frequent data transfers lead to unsustainable power consumption. Single, complex, hyper-pipelined processors are thus replaced by parallel architectures with simpler processors (*multicore architectures*) and/or specialized components (*heterogeneous architectures*). This allows designers to significantly reduce power consumption: a specialized accelerator is tailored to execute specific functions, activated only when needed, and can be turned off to save additional static power when unused [4].

Designing specialized accelerators is complex, expensive, and time consuming. Designers have to determine the proper microarchitecture to achieve the desired performance with minimal resources. The microarchitecture is usually designed for *spatial computation*, i.e., to execute operations on multiple data in parallel. The resulting design must be described with a *hardware description language* (HDL) for enabling the actual hardware implementation. However, this process requires specific hardware design skills that are uncommon in application designers. Finally, implementing a specialized accelerator as an Application-Specific Integrated Circuit (ASIC) allows designers to achieve the best energy efficiency but limits the reusability of the components and, in turn, the sustainability of the architecture design process. Field-Programmable Gate Array (FPGA) devices are becoming attractive to reduce the cost of accelerator development by reusing the same resources across multiple components after *reconfiguring* the device.

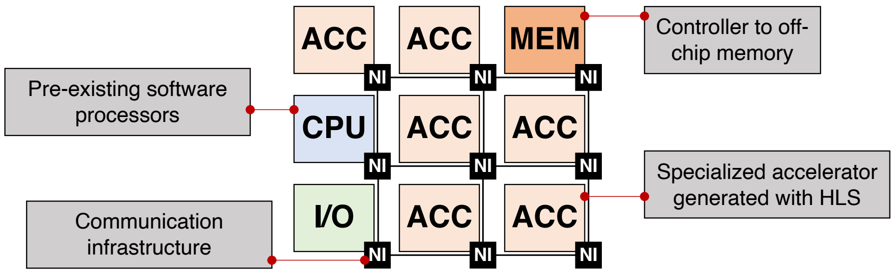

ACC
ACC
MEM
Controller to off-chip memory
Pre-existing software processors
CPU
ACC
ACC
Specialized accelerator generated with HLS
Communication infrastructure
I/O
ACC
ACC
NI

Fig. 1 Example of heterogeneous platform: CPUs and auxiliary elements are predesigned components, while accelerators can be designed with HLS. The FPGA can host the entire system or only some accelerators.

To cope with the increasing complexity of such heterogeneous architectures, designers need to raise the abstraction level for both system and component design. At the system level, custom design flows are replaced by more reusable architectures based on the concept of “platform”, like the one shown in Fig. 1. A platform template is progressively refined to obtain the final architecture with proper customizations [5]. This approach operates at the system level, describing the component functionalities and their interactions above the register transfer level (RTL), and uses automated methods for component synthesis and integration [6]. This enables the reuse of the same high-level components across multiple projects, significantly reducing design time and costs [7]. At the component level, manual design is replaced by automated methods based on *high-level synthesis* (HLS), which is the process of automatically translating a behavioral specification into the corresponding RTL description ready for synthesis [8, 9]. This process is similar to the generation of machine code with compilers. HLS can be divided into three phases: the analysis of the input description to extract relevant information regarding the functionality to implement, the creation of the optimized micro-architecture that implements the given functionality, and the generation of the final RTL descriptions for the subsequent synthesis steps. This requires the introduction of hardware-oriented concepts (like timing, parallelism, and concurrency) that might not be present in the initial description. In addition, the possibility of customizing the target architecture requires a trade-off between performance and use of resources, and the evaluation of the effects of local decisions on the entire design.

This novel boost in *accelerator-rich architectures* is pushed by an increasing number of different computing domains including from datacenters, high-performance computing, reconfigurable embedded systems, and Internet-of-Things devices [10, 11, 12] with stringent requirements that cannot be met either with processor-based architectures or pre-designed components. So, designers must be able to specify the functionality along with its non-functional requirements, derive the final microarchitecture of the components, and create a system that is able to use it with limited changes to the original application. Software and hardware engineers are still speaking different languages with a significant gap even when using HLS [13, 14]. Although the available HLS tools have a similar organization [15], there are several differences in the approaches used for each phase, potentially leading to very different results. Since HLS is mostly application-dependent, it is impossible to determine an optimal flow for every algorithm and computing system. On the contrary, it is crucial to understand algorithms and methods for the different phases to better understand the results and, eventually, how to guide or improve the tools.

This chapter discusses several aspects that leads to create a successful HLS engine, including the specification of the input functionality, the optimization of the microarchitecture, and the system-level integration. It also discusses problems that are still open along with challenges that can be tackled with proper HLS tools. For example, it describes how HLS can enable the creation of more secure components in an era of increasing cyber-attacks [16].

2 Background: Technology and Models

2.1 Target Technology

When selecting the processing elements for an architecture, designers have to face the never-ending challenge of trading off *performance* and *flexibility*, as shown in Fig. 2. While performance is the main optimization goal for many applications, flexibility allows them to reuse the same system for different applications. The tra-

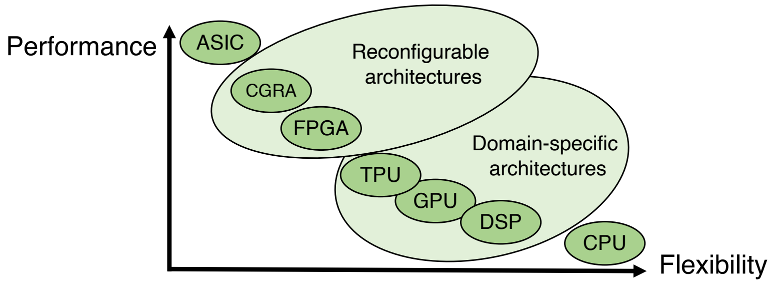


Fig. 2 Trading off performance and flexibility by using different target technologies (adapted from [17]).

ditional **Central Processing Unit** (CPU) is the most flexible component, able to execute any kind of application, sacrificing performance and energy efficiency. A specialized processor, like a **Digital Signal Processor** (DSP), **Graphical Processing Unit** (GPU), or a **Tensor Processing Unit** (TPU), offers better performance for application-specific workloads (e.g., audio, video, or machine-learning applications) while maintaining a certain degree of flexibility. A **Field-Programmable Gate Array** (FPGA) device is an array of elements that can be configured by the user to execute a specific functionality even after fabrication, as shown in Fig. 3. For this reason, the elements are called *configurable logic blocks* and their configuration is called a *bitstream*. FPGAs have the flexibility of processor-like architectures since they can be reused (after reconfiguration) to execute different workloads but they can achieve performance comparable to ASICs thanks to the possibility of implementing specialized microarchitectures on the configurable blocks. Some FPGA devices also offer the possibility of changing their functionality *during* the execution through *partial dynamic reconfiguration*. Designers create all *partial bitstreams* statically (i.e., partial configurations only for the specific FPGA region where the accelerators will be placed). The reconfiguration loads the proper partial bitstream into the corresponding region through a specific port, called an *Internal Configuration Access Port* (ICAP). This process is time-consuming and used only when the benefits of hardware accelerators are much greater than the reconfiguration time. A **Coarse-Grain Reconfigurable Array** (CGRA) is a network of specialized data processing units. Executing an application requires to create only a configuration of the interconnections. It can achieve better performance than FPGA devices but it is more application specific. In an **Application-Specific Integrated Circuit** (ASIC), the microarchitecture of the component is tailored to execute only the given functionality. Specialization allows the circuit to achieve the best performance but limits the reuse and therefore the flexibility of the component. Also, since the functionality of an ASIC cannot be changed after fabrication, manually tuning and verifying the design leads to high design costs. In all cases, it is important to understand how the operations are implemented in hardware to better understand the use of resources. For example, ASIC designs are mapped onto standard cells or hard macro blocks (like memories) and the silicon area is a good metric to characterize the implemen-

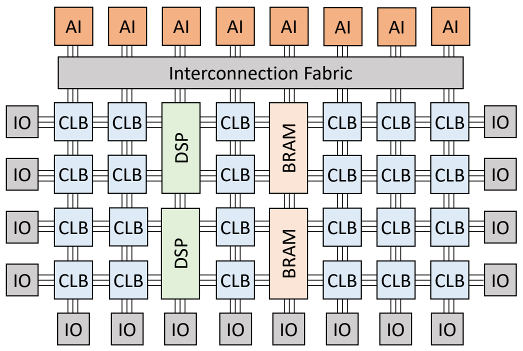


Fig. 3 High-level FPGA organization: The device contains an array of configurable elements and heterogeneous resources (e.g., DSP and Block RAMs). It may also feature dedicated engines that are designed and optimized for AI processing.

tation. On the contrary, FPGA designs can use a heterogeneous set of resources, like configurable logic cells, Block RAMs, and DSP blocks. In this case, comparing two designs is much harder.

2.2 Accelerator Models

Designers have several alternatives for creating the microarchitecture of the specialized accelerators, especially due to different execution modes [18]. *Configurable, extensible processors* have specific accelerators integrated into the pipeline of the given CPU to improve the execution of specific code portions. The selected code to be accelerated is represented with a new instruction and, for this reason, the components are commonly referred to as *custom instruction set extensions* [19]. After selecting the kernel instructions and extending the compiler to target them [20], HLS can create the RTL microarchitecture of the accelerator to be integrated in the processor datapath [21]. Commercial products like **Synopsys ARC** and **Cadence Xtensa** feature complete toolchains to profile the application, identify hot-spots to be accelerated, and integrate the corresponding RTL modules. *Accelerator-rich architectures* feature, instead, several stand-alone components that are designed to provide peak performance for selected large kernels or even complete applications. Large data sets are usually stored in an external memory (e.g., DRAM) that is accessed with specific interfaces. Such accelerators can be *configurable* with parameters that the user can specify (usually through input ports) to select a specific functionality of the component. While components can be extremely complex, many modern applications can be reduced to operations on large streams of data. This is, for example, the case of machine-learning applications that need to perform simple operations (e.g., convolutions) on many data. Control constructs and synchronization can be introduced to reuse the same hardware resources to iteratively operate on new data

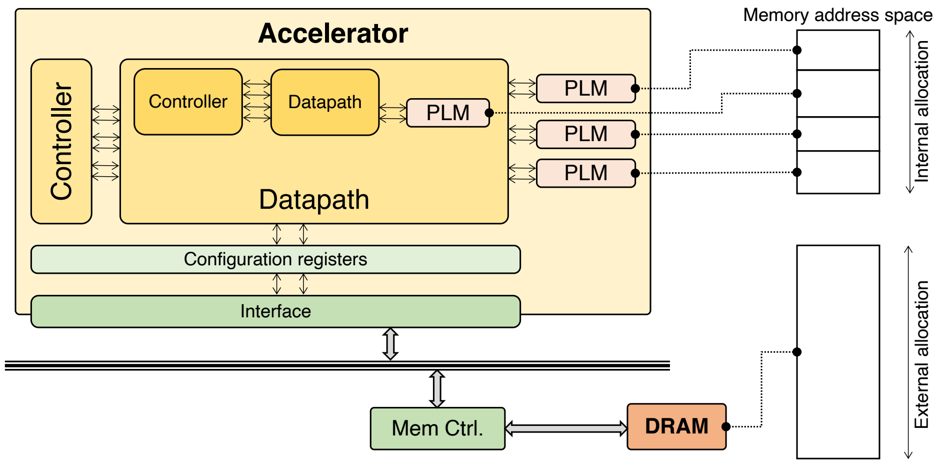


Fig. 4 An accelerator is composed of submodules hierarchically organized, and may contain *private local memories* to store local data and be connected to the external memory.

using the same concepts as software loops. To reduce the complexity of large designs, designers can create *dataflow architectures* as a collection of components communicating with latency-insensitive protocols [22]. The specialization of the components can target the composition of the block [23], the memory architecture [24], or the communication primitives [25]. Modern machine-learning applications benefit from architectures with simple accelerators that are fed with streams of data and reused across multiple cases, like *modular accelerators* for neural networks [26] and *dataflow systolic arrays* for matrix multiplications [27]. For these applications, modern FPGAs are implementing specific vector processor cores, like the AI engines in the Xilinx Versal AI Core Series [28].

2.3 Accelerator Template

Each specialized accelerator is composed of two fundamental blocks: a *datapath* and a *controller* [29]. The **controller** can be modeled as a finite-state machine (FSM) that determines which operations execute in each clock cycle and sends control signals to trigger the operators and the interconnection in the **datapath** to perform the computation. The datapath is a collection of interconnected hardware resources that can execute in parallel (spatial computation). Additional components include local and external memories for data *storage*. Internal memories are accessed directly by the datapath resources, while external ones are accessed through pre-defined components like memory controllers. When the complexity increases, modules are organized in submodules with the same structure, like in software functions. Usually common functions are replicated at different levels of the hierarchy, like uniquification in logic synthesis, although solutions attempt to reuse the same hardware blocks with special proxies to eliminate those function copies and reduce resource requirements [30]. Accelerators are usually encapsulated in an infrastructure to interact with the system, as shown in Fig. 4. They can be also interconnected with each other for direct communication. This architecture is often used for dataflow applications, where data are streamed from one component to the next [31]. The processor core (CPU) executes a software application to prepare the data in memory and configure the accelerator through the interconnection system (e.g., a bus or a network-on-chip) with memory-mapped operations on the **configuration registers** that are connected to the input ports [4]. The data stored in the external memory (DRAM) are accessed through one or more memory controllers [32]. DMA mechanisms allow accelerators to exchange large data blocks with DRAM [18, 24]. When moved inside accelerators, the data are stored in **private local memories** (PLMs) for fast access. PLMs can also store data for the entire execution of the accelerator [33]. Accelerators access PLM data with known latency (e.g., one or two cycles) while the latency of external accesses is usually unpredictable. While PLM accesses make the scheduling of memory operations and the controller creation simpler, accelerators must implement a latency-insensitive memory interfaces to guarantee execution correctness when accessing external data. An FPGA can host the entire system or only the accelerator part. In the latter case, the FPGA is combined with a hard-core processor through an interconnection fabric or is a stand-alone component, like in the IBM *cloudFPGA* project [34]. IP and technology vendors provide intellectual property (IP) blocks for common functions. For example, Synopsys and Cadence offer soft IPs for high-speed communication (e.g., SerDes IPs) like Ethernet physical layers. Similarly, FPGA vendors offer a list of configurable IPs for common peripherals like DMA controllers to exchange data with external memory banks, to access USB ports and PCIe bridges, and to display data through video controllers.

3 Introduction to High-Level Synthesis

The design of specialized hardware accelerators requires a design flow that allows designers to generate the register-transfer level (RTL) microarchitecture associated with the desired functionality. High-level synthesis is a key technology in this context. It automatically translates an input high-level specification into the corresponding RTL implementation ready for logic synthesis (either for ASIC or FPGA technologies). **High-level synthesis** is, indeed, a collection of methods and algorithms to automatically define the RTL micro-architecture of a hardware module starting from the specification of its functionality at a higher level of abstraction. This process is similar to the generation of machine code for programmable processors by compilers.

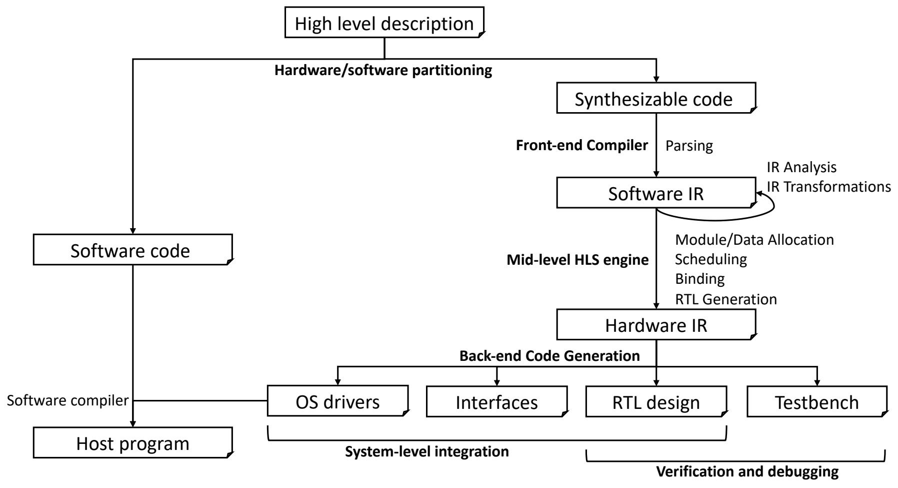


Fig. 5 Overall organization of a classic HLS flow.

3.1 A Traditional High-Level Synthesis Framework

Figure 5 shows the overall organization of a classic HLS flow. The *high-level code* represents the input functionality to implement. The designer can start from an existing algorithm described in software-like languages (e.g., C, C++, or Python), hardware-oriented languages (e.g., SystemC), or domain-specific languages (e.g., Chisel). Each language targets specific application domains or designer cases (see Section 4.1 for more details). First, the input code is split into two parts (**hardware/software partitioning**): the code to be executed by a software processor and the part to be implemented in hardware. The former follows a classic compilation flow to create the binary executables, while the latter must be translated into the corresponding hardware description and integrated with the CPU.

This input description is processed by a front-end to remove language-dependent details and extract the *essential semantics*, which is represented through a more generic *intermediate representation* (*SW IR*). Since the IR impacts the following steps, the front-end phase includes several **IR transformations** to obtain a more hardware-oriented representation.

In the mid-level phase, the core of the HLS engine creates the accelerator micro-architecture, i.e., the list of hardware submodules and their interconnections. The HLS engine is composed of several steps. **Scheduling** defines *when* each operation can start its execution to satisfy dependencies and maximize hardware parallelism. **Allocation** and **binding**, instead, determine *where* and *how* each operation is executed. For example, module allocation determines the number of functional units, while module binding assigns each operation to the proper resource to avoid conflicts. The same applies to data and memories. **Data allocation** includes the definition and specialization of the memory architecture to efficiently store and move the data to reduce bottlenecks, i.e., the situations where computation resources are stalling because data are not available. Finally, **RTL generation** creates a representation of the resulting micro-architecture (*HW IR*), along with the logic to control the execution.

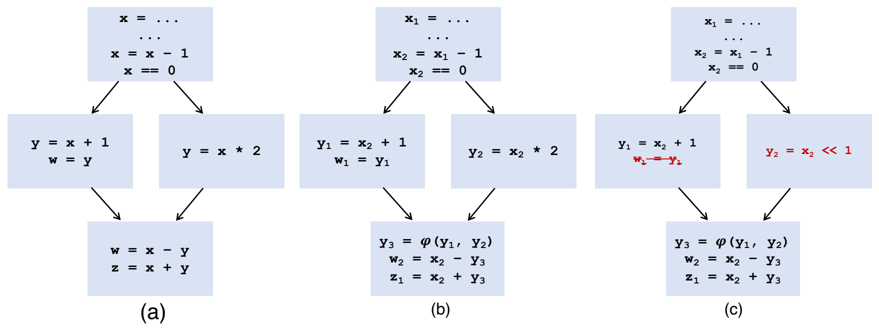
Finally, the back-end phase produces the artifacts for the subsequent design steps. First, **code generation** produces the target RTL description in the desired hardware description language (HDL) (*RTL design*). Similarly, **testbench generation** and **interface generation** produce elements to support system-level integration and verification of both the component and the system, respectively. This entire process can be part of a larger design-space exploration framework to trade-off the different design objectives within the final system.

3.2 A Bit of History on Commercial Products and Academic Projects

The first HLS projects targeted ASIC with so-called *silicon compilers* [35, 36], used especially for simple, data-intensive applications. For example, **Chippe** [36] included layout constraints during behavioral synthesis. As the complexity of the hardware modules started increasing, designers shifted towards high-level languages and compiler-based HLS frameworks [37]. Also, since FPGA devices allow for a fast turn-around time to achieve a solution while ASIC requires extensive fine tuning of the designs, HLS tools have been mostly developed for FPGA [6], with several academic prototypes and commercial solutions [15] (see Chapter on "FPGA-Specific Compilers" for more details on FPGA HLS tools).

Commercial tools are more oriented to a horizontal approach, simplifying coding, porting, and analysis of the solutions. In most of the cases, such tools are offered by FPGA vendors to simplify the use of their devices, in some cases free of charge. For example, **Xilinx Vivado HLS** targets Xilinx FPGA devices and **Intel HLS Compiler** produces RTL code for Intel FPGA devices. Other tools, like **Siemens EDA Catapult HLS** or **Microsemi LegUp HLS Compiler**, are not vendor specific and can target a broader range of devices. Some HLS tools also target ASIC, like **Cadence Stratus**, **Siemens EDA Catapult HLS**, and **NEC CyberWorkBench**. All these tools have graphical user interfaces or TCL scripts to automate the steps, with good estimators for performance and resource usage. Indeed, they are often tightly connected to logic synthesis tools to provide accurate resource characterizations, especially in case of ASIC. Most HLS tools also provide synthesizable libraries of communication and synchronization protocols for building more complex systems by focusing only on computational aspects [25].

Academic projects, instead, are usually more focused on research and experimentation with a vertical approach. For example, **Spark** [38] was the first *public* HLS framework, where the designer could set constraints on the resources to show the relevance and impact of compiler transformations. **GAUT** [39] was a framework for DSP applications. GAUT introduced the concepts of memory mapping, communication modules, and I/O timing to create pipelined architectures. **xPilot** [5] was the first project to provide a complete framework for the synthesis of application-specific con-



(a)
```
x = ...
...
x = x - 1
x == 0

y = x + 1
w = y

y = x * 2

w = x - y
z = x + y
```

(b)
```
x_1 = ...
...
x_2 = x_1 - 1
x_2 == 0

y_1 = x_2 + 1
w_1 = y_1

y_2 = x_2 * 2

y_3 = φ(y_1, y_2)
w_2 = x_2 - y_3
z_1 = x_2 + y_3
```

(c)
```
x_1 = ...
...
x_2 = x_1 - 1
x_2 == 0

y_1 = x_2 + 1
~~w_1 = y_1~~

y_2 = x_2 << 1

y_3 = φ(y_1, y_2)
w_2 = x_2 - y_3
z_1 = x_2 + y_3
```

Fig. 6 Example of code (a), its corresponding static single assignment (SSA) form (b), and the optimized IR after strength reduction and dead-code elimination (c).

figurable processors and heterogeneous multi-core architectures, focusing on FPGA targets. xPilot had been later acquired by Xilinx to become the base of Vivado HLS, becoming one of the most complete and easy-to-use HLS tools on the market. **LegUp** [40] was an open-source HLS framework based on LLVM that allowed the creation of complete SoC architectures for Altera (now Intel) FPGA devices. Thanks to its modular organization, it has been widely used to prototype different types of solutions for HLS problems, like bit-width analysis [41], profiling-driven optimization [42] and the effects of compiler optimizations [43]. It is now discontinued as it became a commercial product, Microsemi LegUp HLS Compiler. **Bambu** [44] is one of the remaining open-source HLS frameworks. It allows designers to experiment with HLS solutions thanks to a modular and dynamic compilation framework based on both GCC and LLVM [45]. It focuses on the problem of understanding how to synthesize C/C++ semantics. To do so, it offers a unique memory microarchitecture that supports most of the C constructs without semantic changes (including dynamic pointer resolution and memory allocation [33]). It has been also used to integrate solutions for hardware and hardware-assisted security, like intrinsic dynamic information flow tracking [46], IP watermarking [47], and algorithm-level obfuscation [48]. There are also projects that focus on specific accelerator models (like dataflow compilers) or application domains (like deep learning), especially for FPGA targets. The interested reader can refer to the Chapter on "FPGA-Specific Compilers" for more details.

4 From Input Specification to Intermediate Representation

This section discusses the transition from the input high-level specification into a language-agnostic representation that is optimized for hardware generation. One of the first major challenges in HLS is the right choice of input language and compiler (with associated transformations) based on the characteristics and requirements of the given application (e.g., the expected latency).

4.1 Input Specification and Intermediate Representation

Most pre-existing algorithms are described with traditional languages (like C, C++, Fortran, etc.). Therefore, modern HLS tools are mostly built on top of state-of-the-art software compilers, like **LLVM** and **GCC** [49, 43]. These compilers have support for many input languages and can support the porting of legacy code into hardware. Also, they can leverage many years of research in compiler construction to extract, analyze, and optimize a representation that is more suitable to hardware implementation. To simplify compiler optimizations, the IR is usually translated into a *static single assignment* (SSA) form where multiple assignments to the same variable create different versions, as shown in Fig. 6. Source-to-source compilers can rewrite existing code into a more hardware-friendly format and expose more "knobs" for optimization [50]. For example, since commercial HLS tools use directives to optimize the input code, source-to-source transformations can automatically insert identifiers (e.g., loop labels) and transform the code to better apply synthesis directives.

Modern machine-learning applications make heavy uses of operations on multi-dimensional arrays, also called "tensors". These applications are often extremely parallel and suitable for hardware acceleration. In such systems, the creation of the memory architecture demands efficient methods to describe the operations and the data access patterns. Many languages have been proposed as the frontend to HLS frameworks to better expose such details. **Halide** [51] simplifies the descriptions of high-performance image and array processing code, while **Halide-HLS** [52] is an extension to target FPGAs. Machine-learning applications are built almost exclusively with Python-based frameworks like **PyTorch**, **Tensorflow**, **Caffe**, etc. Python is a popular language that hides many details from the programmers. Compilers translate Python representations into code that can be processed by HLS tools based on traditional IRs.

Traditional compilers progressively transform the IR into simpler operations that are later mapped on machine instructions. This process loses important information for hardware generation, like information of the size of the data structures and the patterns across operations that help define the memory system. Designers are working to extend these representations to pass more information over the compiler passes until it reaches the HLS flow. For example, Google recently proposed LLVM **Multi-Level Intermediate Representation** (MLIR) [53] to create a customizable compilation framework that provides information at different levels. Similarly, **Heterogeneous Parallel Virtual Machine** (HPVM) [54] is a representation for a parallel compiler that aims at simplifying the code implementation of parallel hardware. These representations are often combined with domain-specific languages where the designer can abstract specific hardware details. For example, **Spatial** [55] is a recent language to describe hardware accelerators at a higher level. Such descriptions are later compiled and translated into **Chisel** and then into Verilog. These frameworks can be considered more as "hardware generators" rather than complete HLS tools. Indeed, they operate more as "translators" from the input to the output descriptions, with limited optimizations.

4.2 Analysis and Optimization of the Intermediate Representation

The next phase analyzes and transforms the IR extracted from the source code to create a more hardware-friendly representation and optimize the component to generate. Applying IR-level transformations simplifies the following HLS steps, improving the accelerator's performance or reducing its hardware cost. Some transformations are borrowed from traditional compiler optimizations, while others are specific for hardware. This is another motivation for which it is convenient to base HLS on stable and mature compiler frameworks.

Constant propagation and **strength reduction** are classic compiler transformations that can simplify or even eliminate arithmetic operations in the code. For example, the instruction "`x_2 * 2`" of Fig. 6 can be transformed into "x_2 « 1" (see Fig. 6(c)): a left shifter is much more hardware efficient than a multiplier. Designers may replace some variables with constants representing their average values to leverage these optimizations. These transformations are usually referred as **software approximation techniques** and allow designers to obtain efficient hardware despite minor errors in the results. **Dead-code elimination** removes unnecessary code, which will be otherwise translated into unnecessary hardware (see, for example, instruction "`w_1 = y_1`" in Fig. 6(c)). This applies, for example, when control code depends on input parameters that, in specific accelerator instances, are always set to constant values. HLS is often limited by control constructs, like `if-then-else` statements. Operations in the `true`/`false` branches cannot start their execution until the condition is evaluated. **Code speculation** moves some operations before the condition evaluation so that they can be executed in parallel [56]. Results are temporarily stored in registers and, after evaluating the condition, the values of the "wrong" operations are discarded. This optimization increases the available parallelism, leading to better performance.

Compiler analyses and transformations can also reduce data dependencies and increase hardware parallelism. For example, pointers are widely used to create efficient software code. However, their implementation in hardware is complex because a pointer-based operation must be connected to all the memory locations (either internal or external) where the corresponding information is potentially stored. **Alias analysis** helps determine if two pointers in the source code can ever refer to the same memory location. If it can be proven that two pointers never refer to the same object, there is no dependency, enabling more memory optimizations. **Static pointer resolution** determines the exact variable accessed by a pointer operation, eliminating the need for an explicit pointer. This information can be later used to optimize the creation of the memory architecture (see Section 5) because the two operations can potentially run in parallel when proved to access different data [33]. Additional **memory analyses** determine the list of data structures to allocate in memory and their characteristics (e.g., size and bit-width) for determining whether the corresponding memories fit inside the area constraints of the accelerators. Other transformations operate on the data structures to expose more parallelism. Indeed, arrays are generally stored in memories with limited ports. Designers can apply

array partitioning and **scalar replacement of aggregates** to reduce the number of memory dependencies.

While software developers can overestimate the bit requirements for some data, many applications do not use the full range of the corresponding variables. Since a processor's hardware is already built with pre-defined registers and arithmetic-logic units, the execution with overestimated variables has almost no extra cost. Hardware specialization requires, instead, to determine the minimal resources needed for the computation to trim unnecessary logic. Therefore, the front-end phase also performs **bit-width analysis** to determine the required precision of each operation to maintain execution correctness and **bit-width transformations** to propagate the information through the design with iterative methods. Similar transformations include also **numerical conversions** (e.g., from floating-point to fixed-point representations) to reduce hardware cost but maintain a certain level of accuracy of the results. In this context, many HLS tools, for example *Xilinx Vivado HLS* and *Cadence Stratus*, allow library extensions to manually specify the precision of input/output data or to specify particular bit-level operations on the signals. This feature is particularly useful when HLS is used to generate components to be integrated in larger specialized systems like industrial machines. Synthesizable C++ libraries, like **HLSLibs** have been proposed to extend existing HLS tools with custom precision.

Especially in data-intensive applications, loops account for most of the accelerator execution and it is complex to extract parallelism from their representations. Most of the parallelism is often between consecutive iterations. While spatial execution can create multiple parallel instances of the loop body, hardware synthesis is usually limited by the loop boundaries. Therefore, loop transformations are widely used to expose more hardware parallelism. For example, **loop unrolling** replicates multiple instances of a loop body to execute in the same iteration. Therefore, the number of operations between control branches is increased, potentially leading to more parallelism. However, this transformation requires a careful analysis of the dependencies; otherwise the multiple iterations in the same loop body are serialized without an effective speed-up. *Artificial dependencies* can also be due to conflict on resources, like in the case of limited memory ports, forcing the serialization of the operations. Another important transformation is **loop pipelining**: consecutive loop iterations are partially overlapped. This optimization follows the same principles of instruction execution in pipelined processors, when an instruction can start before the termination of the previous one. In this case, the *initiation interval* (II) is an important parameter: it represents the number of cycles required by the loop to start a new iteration. A perfect pipeline starts a new iteration after each cycle ($II = 1$). In case of large data sets, **loop vectorization** aims at executing operations on multiple data in parallel, increasing the demand of memory bandwidth. Other transformations like **loop fusion** and **loop switching** optimize consecutive loops. The interested reader can refer to [57] for more details on these loop-related HLS transformations.

Most of the information extracted in this phase is passed to the next steps as extra annotations or directly embedded into the IR. These transformations aim at generating optimized hardware but are often strictly interdependent. For example, loop transformations are not able to expose much parallelism if not properly supported by memory optimizations. Similarly, constant propagation can enable further dead code elimination. Some HLS tools, for example *Bambu*, implement *dynamic compilation flows* that re-execute passes whose outcome invalidates previous results or activates further optimizations [45].

5 Creation of the Microarchitecture

To create the microarchitecture of the accelerators, the HLS middle-end requires both the temporal and spatial distribution of the operations to be determined. In the former case, HLS determines *when* to execute each operations, i.e. in which clock cycle, to satisfy any dependency. In the latter case, HLS determines *where* to execute the operations, i.e. on which hardware resources, to minimize the hardware cost while avoiding any conflict.

5.1 Scheduling and Performance Optimization

After defining and optimizing the intermediate representation of the functionality to synthesize, the first step is to determine a set of available resources and introduce the concept of *timing*. This process is highly dependent on the previous compiler phase, so they are often executed in an iterative way until the designer reaches a good trade-off between latency and resource usage. **Allocation** determines how many resources will be used for the given component. Since operations executed in the same clock cycle require different resources for the execution, allocation can limit the number of operations that can execute in parallel, i.e., in each clock cycle. **Scheduling** assigns operations to the clock cycles to balance performance and resource usage. Also, scheduling must take into account technology-dependent information like the latency of the hardware modules where the operations are assigned for execution. Temporal assignment must also respect several types of dependencies among operations. HLS dependencies include *real dependencies*, like data dependencies, but also *artificial dependencies* that do not carry real values but are needed for the correct execution of the specification. The result of an operation must be served to the following operation (if feasible in the given clock period) or stored in a register for use in the next clock cycle. This requires the latency of the circuit to be analyzed and compared with the **clock period** (the available timing budget for each synchronous event), i.e. by estimating the **slack** of each clock cycle [58]. The slack is the margin between the delay of the circuit and the given timing requirement. Negative slack means that the timing constraint is violated, while a positive slack means that an extra delay could be tolerated. There are multiple situations during scheduling:

- the operation terminates much earlier than the clock period (i.e., the *slack* is positive and high); in this case, it might be possible to execute another operation provided that it fits in the remaining time (**operation chaining**);

- the operation terminates right before the end of the clock period; the result will be then used in a subsequent cycle and it must be stored in a register;
- the operation takes more than one cycle and its result shall be saved only when finished (**multi-cycling**); in this case, if the functional unit is pipelined (i.e., it contains internal register to create computational stages), another operation can start on the same resource even before the current one is completed.

Scheduling is an NP-complete problem and considers different aspects to generate a *valid* implementation (i.e., a circuit that does not produce computational errors). It is usually applied at the basic block[1] level to extract more parallelism. Each operation must start its execution only after its predecessors have produced the corresponding results. Then, there must be a physical resource that is able to implement the given operation and is not already in use. So, common HLS tools use heuristic methods to obtain efficient solutions in a reasonable amount of time. Common scheduling algorithms include **list-based scheduling** [59], which maintains a list of "ready" operations and progressively assigns them to the clock cycles, and **system of difference constraints (SDC) scheduling** [60, 61], which operates on a rich set of scheduling constraints. List-based scheduling is simpler, faster, and more efficient on pure datapath descriptions, while SDC scheduling achieves better results for loop descriptions [56].

Exact methods are still applied in case of control-based designs, where the scheduling problem is combined with code motion to execute part of the function speculatively even before a control condition is evaluated (code speculation) [62]. An alternative approach includes the use of exploration algorithms (e.g., genetic algorithms and particle swarm optimization) to evaluate a variety of solutions. These methods are more efficient since they can find a combination that is more specific for the application but are time-consuming [63].

Executing memory operations in the same clock cycle requires that such operations are independent and there are no conflicts when accessing the data. Conflicts can be avoided by accessing different memory resources or by having memory resources with multiple ports. However, the latency of memory operations depends on where the data are allocated. Accesses to local data have fixed latency (one or two cycles depending on the latency of the memory) and the corresponding memory operations can be considered as other operations. When the data are allocated outside the accelerators, the HLS engine must follow safe assumptions to guarantee correct execution in all cases: the corresponding data could be allocated off-chip or multiple accelerators can access the same memory creating contention and additional latency. Therefore, the scheduling algorithm must assume the external memory operations have unknown latency [64]. The same case applies to operations corresponding to unpredictable components, like data-dependent submodules. In case of operations with unknown latency, the scheduling uses **latency-insensitive protocols** [22] to guarantee that the computation proceeds only when the operation is completed. However, executing multiple operations with variable latency in the same clock cy-

[1] A *basic block* is a sequence of instructions with a single entry-point and no internal branches. The basic block definition is induced by the control constructs contained in the code.

cle is complex. Therefore, these operations are generally serialized by most of the HLS engines. This serialization may become inefficient when trying to optimize the latency or guarantee the worst-case execution time. Designers proposed approaches for dynamically scheduling the operations. While this concept is highly efficient, the area overhead for implementing the control logic is high and the dynamic scheduling is usually limited to specific cases, like memory-related operations [65, 66].

5.2 Binding and Resource Optimization

After introducing timing inside a functional specification, it is necessary to determine which physical resources are effectively used in the target microarchitecture. The following **binding** phases aim at reducing the amount of hardware modules by defining the possibilities of *resource sharing* [67]. This process includes also the definition of the memory architecture, including the partitioning of the data. However, this aspect is discussed in Section 5.3. The binding phase includes the following steps:

- *functional-unit binding* defines which functional unit is used to execute each operation of the specification. It must ensure that each operation is assigned to a functional unit without conflicts, i.e., different operations are not executed by the same functional unit in the same clock cycle.
- *register allocation and binding* determine which data values cross the cycle boundaries and must be stored locally into temporary registers. It also determines how many physical registers must be effectively used (exploiting reuse whenever possible) and how the data values are assigned to them [68].
- *interconnection binding* determines how to connect the datapath resources, which resources are needed to multiplex the signals in case of different paths coming to the same input port of a resource (either functional unit or register), and the corresponding control signals to generate in each clock cycle to correctly route the signals based on the operations to execute.

Several aspects may impact the final implementation, demanding specific approaches. For example, in case of process variation, scheduling and binding must be considered together with statistical approaches. Execution paths in the specification may have a different probability of execution, and most-executed paths could be more optimized in terms of latency and resource usage. Operations executing in the same clock cycle require distinct functional units to avoid conflicts. Conversely, operations executing in different clock cycles are *compatible* and, if they are of the same type, they can share the same functional units to reduce resource usage. The *resource binding* problem requires the definition of this set of compatibilities.

Register binding is similar to the process of assigning temporary values to processor registers. The main difference is that, in HLS, there is the possibility of customizing the microarchitecture to add more registers when needed. So, register spilling is not necessary. However, it is still necessary to compute the *liveness* of each variable to determine when different values are compatible and can share the same register. Liveness analysis determines in which points each variable contains a valid value that must be stored in a register. The liveness of a variable v is defined as the interval of time (and, thus, the corresponding clock cycles) between the definition of v and its final use. This interval defines the time for which the value must be stored in a register. Given this definition, two variables u and v are *compatible* if their liveness times are not overlapping. Compatible variables can be stored in the same register because there will be never a moment when both values are needed. Liveness analysis is thus a critical step in register binding. In SSA-based representations, each version can be considered a new variable and the liveness intervals of these new variables have no interruptions. The register binding can be defined more easily and efficiently. Register binding must also consider the technology for implementing the registers and their impact on the final power consumption of the circuit. For this reason, architectures with multiple supply voltages have been proposed and register binding has been adapted for these cases. Special registers, like razor flip-flops [69], are used to tolerate variations in the latency of the operations to avoid timing violations.

Both these problems are usually represented with a **compatibility graph**. Two operations or two values are compatible (i.e., they are connected with a compatibility edge) when the following two conditions are verified: 1) the two operations can be executed by the same functional unit or the two variables can be stored in the same register, and 2) there are no timing conflicts. A compatibility graph is dual to a **conflict graph**, and the approach to generate them must be conservative to guarantee correctness in every circumstance. Analyses and transformations can remove an edge from a conflict graph or add an edge to a compatibility graph when the property holds in every case. A compatibility problem described with a compatibility graph can be easily solved with *clique covering* formulations, while the dual conflict problem is solved with *coloring* formulations.

This phase must also consider the effects of scheduling on all combined resource binding problems. For example, executing multiple operations in parallel improves the execution time but usually requires more functional units (since operations cannot share the same resources) and registers (since multiple values are produced in the same clock cycle). In addition, resource sharing usually creates multiple paths to the input ports of both functional units (since different operations may require values from different sources) and registers (since different variables could be produced by distinct functional units). When a port receives signals from multiple sources, HLS engines introduce multiplexers to determine the path active at any given time and drive the signal values. The controller FSM generates control signals to activate the proper paths from source to destination in each clock cycle (see Section 5.4). Resource binding has a huge impact on the number of multiplexers to add. Several methods have been proposed to consider the impact of interconnections during HLS. For example, register binding can be combined with **port swapping** [70]. This optimization swaps the inputs of commutative operations, aiming at reducing the number of paths to each port of the units. This reduces, in turn, the number of multiplexers that are needed.

5.3 Definition of the Memory Architecture

Nowadays, memory optimization is one of the most important aspects of accelerator design since these components need to process a huge amount of data. However, the accelerators can store on chip only a limited amount of data, usually orders of magnitude less than the total amount. When accelerators operate on more data than can be stored on-chip, they also need to interface with an external memory (e.g., DRAM). This must be taken into account when defining the microarchitecture to avoid executing operations when the data are not available yet but also to hide the communication latency. Latency-insensitive protocols in the interfaces guarantee correct execution in case of unpredictable latency. *Multiple memory channels* and *ping-pong buffers* can hide the latency in accessing the data by parallelizing the data transfers and partially overlapping computation and communication. In case of predictable computation, *pre-fetchers* can anticipate data transfers. These solutions are especially applied in specific application domains, like DNN accelerators, where the structure of the layers can provide information to optimize the architectures. For example, having information on the layers allows optimizations to share buffers and reduce resource utilization with liveness and time span analysis.

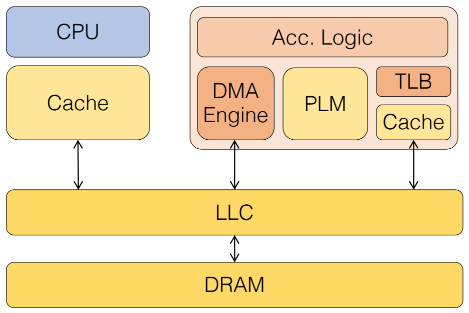
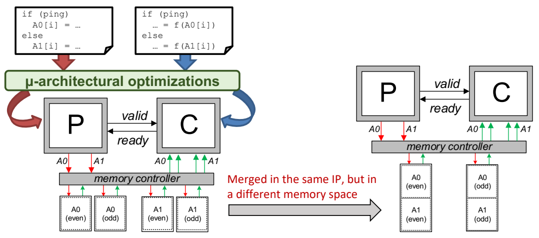


Fig. 7 Memory architecture for specialized accelerators: It can feature caches (with the same principles as CPUs) and private local memories (for fast and deterministic accesses). The elements can share a last level of cache for better performance.

The huge latency from external memory access can be mitigated using specialized memory architectures inside the accelerators, such as **caches or private local memories**, as shown in Fig. 7. These components, however, must be co-designed accordingly with the algorithms and optimizing the corresponding architectures requires a in-depth analysis of the memory behavior of the application. For example, solutions have been proposed to include specialized caches where each array of the input specification that is mapped to external memory has its own cache. But the designers need to apply them only to accesses that can guarantee a certain degree of temporal and spatial locality. In this way, the accesses to different arrays are executed in parallel and the caches mitigate the memory access latency as the memory hierarchy in traditional CPUs. A private local memory is a memory that resides on chip and offers fixed-latency data accesses since it has no logic to support misses. Critical or frequently-accessed data can be placed in private local memories to ensure low latency access. The memory behavior of an algorithm is application-dependent, ranging from statically predictable patterns (e.g., stencils for multimedia applications) to irregular memory accesses (e.g., pointer chasing for graph analytics). Compiler transformations based on **polyhedral models** can optimize the memory accesses in case of predicable patterns. Representing operations with polyhedral models allows designers to apply *affine transformations* to make the iterations independent and extract more hardware parallelism. However, to provide enough memory bandwidth, these transformations must be combined with proper **multi-port memory architectures**. Memory IP blocks, like Block RAMs for FPGA or Static RAMs for ASIC, have a limited number of ports. Such architectures provide the requested data in a fixed amount of cycles when the accesses are distributed to different ports, i.e. there are no conflicts. By partitioning memory into separate physical memories, more data can be accessed simultaneously as more ports are available (*memory banking* – see Chapter on “FPGA-Specific Compilers” for more details). The designer must decide how to partition the data structures at design time and determine where to store each of them. This is usually a trade-off between predictable memory accesses, which are possible when the given structure is stored in the local PLM, and size of the accelerator, which can be reduced by storing more data structures in DRAM. Then, the design of the private local memories of an accelerator comes with many other decisions such as how many memory blocks, what sizes should the blocks be, how can they be arranged to reduce resource usage, and what bandwidth should be used. It is time consuming for designers to make these decisions manually, so they increasingly rely on automated design space exploration methods to predict or estimate the effects of their decisions. For example, the *gem5-Aladdin* simulator can be used to explore the design space and inform many memory design choices, such as to use private local memories, scratch-pads with DMA or caches, or the local memory size and bandwidth. FPGA prototyping with *ESP* can be used to explore and evaluate full systems before ASIC implementations [4].

The cost of memory elements has a significant impact on hardware resources, limiting the amount of data that can be stored on-chip. However, if two data structures have non-overlapping lifetimes, the physical memory banks used to store these structures can be shared as they will be never accessed concurrently. Similarly, if it can be guaranteed that two data structures, while they may exist at the same time, are never written concurrently or read concurrently, these structures can be placed in the same memory bank (but in different memory spaces) and there will not be port contention. This memory bank sharing can greatly reduce the resource usage. *Mnemosyne* is a tool designed to generate an optimized memory architecture based on a set of characteristics of the data structures and the access patterns that can be provided by the designers [24]. The tool analyzes such compatibilities and applies automatic technology-aware transformations (supporting both FPGA and ASIC technologies) to select the proper physical banks in the given technology and determines how to share these physical banks when the data structures mapped on them have disjoint lifetimes. Mnemosyne encapsulates the memory banks in



```
if (ping)
  A0[i] = …
else
  A1[i] = …
```

```
if (ping)
  … = f(A0[i])
else
  … = f(A1[i])
```

µ-architectural optimizations
P
valid
ready
C
A0
A1
memory controller
A0 (even)
A0 (odd)
A1 (even)
A1 (odd)
Merged in the same IP, but in a different memory space

Fig. 8 Resource optimizations for memory IPs with *Mnemosyne* [24].

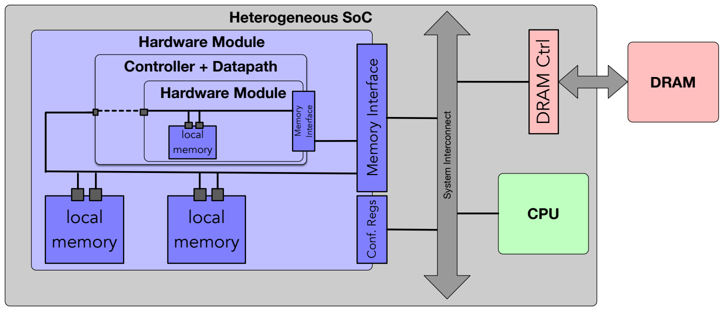


Fig. 9 Daisy-chain memory architecture to support the dynamic resolution of memory addresses in *Bambu* [33, 44].

lightweight memory interfaces that translate the logical requests to the data structures into physical requests to the generated bank configuration (see Fig. 8)z.

Irregular memory accesses are often data dependent or require dynamic resolution of the pointers. These operations are complex when executed in hardware because of the limited flexibility and only few HLS tools support memory architectures with **dynamic pointer resolution**. For example, *Bambu* builds a daisy-chain architecture on top of alias-analysis results, as shown in Fig. 9. The memory allocation step *resolves* the pointers, i.e. converts them into classic memory operations, when the set of possible target data structures is limited to one. Such operations are later directly connected to memory that stores the corresponding data structure, potentially increasing the parallelism on memory operations. Operations with pointers that cannot be "resolved" are connected in daisy-chain with the potential memories. At design time, the HLS engine assigns a specific memory address to every data structure and, in turn, the associated memory space both on-chip and off-chip. At run time, when an address is propagated through the daisy-chain, only one memory will be activated by the request. In particular, when the address refers to data allocated off-chip, the request reaches the external memory interface and is sent to the corresponding memory controller. The scheduling phase can further distribute the memory accesses to hide the latency of memory transfers especially when accessing the data off-chip.

Latency-insensitive protocols enable the creation of **latency-tolerant architectures**, called *elastic circuits* [71] or *dynamically-scheduled architectures* [72]. These architectures can parallelize memory accesses in a way similar to CPU out-of-order execution of the instructions [73] but may have high hardware costs.

Multithreaded software using libraries such as Pthreads or OpenMP can be synthesized into parallel hardware [74]. However parallel hardware often attempts to access the same memory at the same time, resulting in contention and delay. To hide the latency of memory requests and reduce the resource usage of the “hardware threads”, designers can interleave the hardware execution of different functions as done in software threads when resources are busy [75]. Each hardware thread can have its own private memory with a banked architecture.

Memory allocation techniques can map data onto private local memories and caches based on the dynamic memory requirements. Designers propose **overlay** components to increase the flexibility of the FPGA-based systems. For example, *Intel* recently proposed an FPGA overlay for Neural Network (NN) inference that can support different NN architectures without reconfiguring the logic cells but changing only the configuration of the component. Supporting these components requires additional HLS compilation steps to generate the overlay configurations as a set of software-like microinstructions. These configurable layers can also help isolate the computation and the outstanding memory requests for security reasons.

Other flexible approaches include **systolic array architectures**. These architectures are becoming popular for deep neural networks thanks to their flexibility and scalability. *Gemmini* is a representative example of such architectures [76]. It includes an array of “simple” *processing elements*, which perform MAC operations and rounding bitshifts. The PE array can be configured statically or dynamically to execute different dataflows. The surrounding memory architecture enables full utilization of the MAC units and can be customized with information coming from HLS. When performing operations on the PE array, the data are moved from the main memory to the on-chip memories, and the array is configured to execute the given dataflow. Ping-pong buffers are used to overlap computation and communication. The same optimizations described above (multi-port memories and distribution of the accesses to avoid conflicts) can be applied to optimize the architecture. Designers must explore similar trade-offs between more banks for higher throughput and fewer banks for better wiring and physical constraints.

5.4 Creation of the FSM Controller

After defining the complete microarchitecture of the accelerator, the HLS engine must define the control part, i.e. the component that determines the control signals for the datapath in each clock cycle. This part is modeled as a deterministic **finite-state machine** (FSM). An FSM is a directed graph where each node represents a *control state* and the edges are the transitions from one state to another. Each control state contains the set of operations to be executed in the corresponding clock cycle, along with the control signals for the datapath resources (e.g., multiplexer selectors and register write-enable signals). For each operation to execute on a given functional unit, the FSM determines the paths for providing the input values to the functional unit and the results to the next resource (either another functional unit in chaining or a target register). Based on the scheduling and the evaluation of the conditions in the datapath, the control state determines which transition to activate, i.e. which is the next control state to execute.

The controller FSM manages not only the execution inside the accelerator but also the **system-level synchronization** with external components. This aspect is relevant especially for control-dominated applications (where the accelerator's execution can vary based on external signals) and streaming architectures (where data availability can change the accelerator dynamics). In these cases, the controller must receive additional control signals from the rest of the system to determine which specific transitions must be activated inside the FSM.

6 RTL Generation and System Integration

After the HLS engine defines the microarchitecture of the accelerator (both datapath and controller), the last phase produces the HDL descriptions for the subsequent logic synthesis and the auxiliary files for system-level integration.

6.1 Code Generation, Evaluation, and Verification

Code generation produces common HDL languages like Verilog, SystemVerilog, and VHDL. Some components may require **target-dependent descriptions** to be compatible with synthesis tools. For example, *Bambu* generates slightly different descriptions of the internal memories when targeting different FPGA vendors. FPGA synthesis tools may infer these components in different ways. Similarly, targeting ASIC technologies requires to instantiate the vendor-specific descriptions of the proper Static RAMs. For example, *Mnemosyne* provides an abstraction to low-level details. The designer is only required to provide a wrapper around the specific SRAMs to match the standardized signal descriptions.

In this phase, many tools (e.g., *Xilinx Vivado HLS* and *Bambu*) provide **early estimations** on the hardware cost of the design. While complete synthesis can provide more accurate results, this process is time-consuming and not feasible when the designer needs to explore many alternatives before finding the most favorable implementation. These estimations are based on different methods, including cumulative costs of single resources, linear regressions, and graph neural networks to include feedback from actual synthesis steps [77].

Once the design has been finalized, the designers also need to verify that the hardware execution produces the expected results. Indeed, errors can be caused by incorrect language specifications, misuse of synthesis directives, wrong connection of the components, or bugs in the HLS tools. While formal verification is a well-established step in logic and physical synthesis, this approach requires reference and current designs to be described in hardware languages. However, in case of HLS, the input code is usually a software-level, untimed specification. Therefore, simulation-based approaches remain the preferred solution for **HLS verification** [78]. Heterogeneous architectures exacerbate these verification issues for the application programmers, especially when the computation is distributed across software and hardware tasks, and the accelerators are generated with different methods (e.g,. pre-existing IPs, manually-designed components, and HLS modules created with different toolchains). Hardware/software debugging allows designers to backtrack the origin of a bug to the exact point of failure. Since the complexity of architectures is increasing, bugs could be exercised after a long execution time. Therefore, in case of system-level verification, simulation-based approaches have been progressively replaced by on-chip debugging. Many FPGA vendors, for examples, provide automated methods to trace internal signals, exposing them to the user to identify execution anomalies. Advanced methods automatically identify discrepancies between hardware execution and the expected behavior (precomputed in software), restricting the area where the error originated [79]. These approaches are particularly efficient in FPGA thanks to the reconfiguration of the logic. The designer can implement a design with on-chip monitors, execute it directly on the target system to verify the behavior, and modify the functionality in case of problems. For this reason, **FPGA prototyping** is largely used also in ASIC design flows to emulate the functionality of the entire chip and test the interaction with the software, including the Operating System (OS) [80].

6.2 System-Level Integration and Optimization

The HLS-generated components require integration with other **pre-existing components** and the rest of the system. For example, accelerators that interact with off-chip memory require an interface to the **physical memory controller**. Many vendors provide synthesizable IPs that hide the specific details of the controllers, exposing a standard interface to the accelerators. Such interfaces are usually based on standardized protocols, like AXI4, OpenCAPI, or Wishbone.

Hardware/software integration requires the proper **software stack** to be designed to invoke the accelerator from the software code and exchange data with it. The ESP platform is a paradigmatic example of seamless integration of HLS-generated components with a standard Operating System (OS) [4]. From the application viewpoint, the user interacts with the accelerator through two functions for memory allocation and deallocation, and one function to start the accelerator's execution. At a lower level, the interaction is based on standard OS device drivers that are automatically generated during the accelerator design flow. The drivers perform I/O operations on memory-mapped registers to configure the input ports of the accelerators and control the execution. An accelerator's termination signal is connected to a standard interrupt line. When the accelerator completes its execution, it triggers the interrupt routine that reads the output results from the shared memory or the I/O registers.

This software stack simplifies the integration of accelerators, minimizing the changes required to the original application.

Domain-specific libraries can provide abstractions to efficiently create FPGA systems. For example, *Xilinx Vitis AI* provides optimized IP cores, tools, libraries, and models to deploy AI algorithms on the Xilinx FPGA devices. Such abstractions include interfaces to machine-learning frameworks like *Caffe* and *TensorFlow*, internal model representations for efficient HLS implementations, specific AI optimizations (e.g., model quantization), runtime libraries, and configuration overlays for the Deep Learning Processing Unit (DPU).[2]

7 Open and Modern Challenges

This section discusses open challenges that still need to be addressed in HLS and modern challenges in hardware design that can be efficiently addressed and tackled with the support of HLS.

7.1 Creation of Domain-Specific Architectures

So far, most of the optimizations focused on improving the computational part of an accelerator to extract more parallelism and reduce the execution latency, leaving the memory latency to dominate in the overall execution time. DRAM speed is not increasing with the cumulative speed of all components (especially in the case of parallel and/or heterogeneous computing), leading to the so-called **memory wall**: the overall performance becomes limited by the latency of the memory accesses. To address this **memory bandwidth bottleneck**, specialized and carefully-crafted memory architectures can help to hide or reduce the latency when accessing the data. However, this process is complex as it requires several details that are currently lost in the compilation process. Software programmers usually need little to no thought to be put into memory management, where hardware optimizations (e.g., complex cache hierarchies, bypassing, etc.) hide the latency. In case of accelerators, the specialization of the memory architectures adds effort to the development time, both at the software and hardware levels [81]. At the *software level*, algorithms need to be reworked to reap the full benefits of a custom memory architecture. Domain-specific languages, like Spatial, incorporate hardware abstractions to ease the description of common memory operations and speed up the development of these accelerators. For example, operations on arrays, tensors, and matrices are common in scientific and machine-learning applications. Such operations must be translated into efficient operations regardless of the allocation of the data and the location/implementation of the memories. At the *hardware level*, deciding how to

[2] The DPU is a configurable engine optimized for convolutional neural network modules.

partition a software data structure requires many considerations. On one hand, larger memories are slower, use more power, and use more resources. On the other hand, the external routing and address calculation logic for a collection of many smaller memories have the same issues. A middle ground between one large memory and many small memories exists which minimizes these issues, but finding the right compromise can be difficult. Automated high-level synthesis can be useful to perform rapid design space exploration to find a desired trade-off point. Specialized modules for pre-fetching the data from external memories can hide the communication latency but require global information on the use of the data. In case of systolic array architectures, the designers have to carefully define the local memories and the buffers between them and the processing elements to also coordinate the data transfers with the off-chip memory. However, this process requires more information than is available in current HLS flows.

Emerging technologies will play a key role in the design of efficient, secure, and reliable accelerators. First, integrated voltage regulators will enable fine-grained power management with dynamic voltage-frequency scaling, while dual-rail memories will help reduce the static power consumption. In addition, some applications, like graph analytics, require frequent but irregular memory accesses to off-chip memory, whose I/O circuitry and refresh activities are responsible for up to 30% of the system energy consumption (**power wall**). The specialization of memory architectures, in this case, must include **emerging memory technologies**. However, many novel technologies, like Hybrid Memory Cube and High Bandwidth Memory technologies are 3D solutions that can store only few gigabytes of data while DDR4 can store up to hundreds of gigabytes (**capacity wall**). Architectures that combine DRAM with *non-volatile memory technologies* (NVM), which require no refresh, can store up to terabytes of data but require a careful co-design that involves the entire stack [82].

Finally, all specializations require modifications to the HLS input languages to embed more domain-specific information. Domain specific languages (DSLs) will be used by application engineers to specify the algorithmic core of the computation. They will be increasingly used to provide rich information to the compiler and lower-level tools about the high-level semantics of the algorithms. While DSLs can help describe functional requirements (like the operations among data structures), additional annotations allow designers to express non-functional requirements or constraints. For example, *Bambu* already supports the specification of custom data allocation via XML files to specialize the creation of the accelerator's memory architecture and the simplification of the logic to compute the addresses. However, a co-design of the accelerators with the algorithm description would allow the automatic configurations of these steps (see Section 7.2). However, DSLs are usually hard to be accepted by software programmers and they may create integration issues. An interesting approach is to use DSL only for specific application kernels embedded in traditional languages, e.g., with template meta-programming in C++, or for high-level specifications of specific workloads (e.g., machine learning algorithms)—See Chapter on "FPGA-Specific Compilers" for more details.

7.2 Programmability and System-Level Optimization

Hardware/software integration has open challenges for **data allocation** in the case of large data sets. Current solutions allow users to allocate data at the software side and transparently access them from the accelerators; these solutions are processor-centric. Data are allocated in the memory hierarchy of the processors (see, for example, the Intel HARP prototype system) or the accelerator requires efficient methods not only to manage the data locally but also to hide the latency to access them. The design of domain-specific accelerators requires the approach to be changed and the components to be co-designed with the domain-specific memory architecture and the corresponding allocation policy. To do so, designers need a unified representation of the software code and the HLS specification to apply holistic transformations at both sides. Modern compiler representations, like MLIR [83], are gaining attraction but require specific customization within HLS frameworks. These representations will enable and ease the integration of specific **memory-specific transformations** in the front-end phase. For example, loop transformations will be coordinated with transformations in the data structures to improve the data accesses based on the technology of the memories and the location of the data. Rich information about the data access patterns of the algorithm allows the compiler to extract more parallelism and embed more intelligence in the memory controllers.

Finally, **design space exploration** will become more and more important to offer alternative solutions to the designers. Indeed, a fully-automated approach is almost impossible to achieve also because the effects of compiler optimizations and transformations are often application dependent. Therefore, HLS tools have limited view or control on the synthesis process and no well-established flows or sequence of passes. Designers must apply transformations, analyze and understand the results, and derive knowledge for further optimization. This process is the same as in software compilers for code optimization, where machine-learning approaches have been proposed for compiler autotuning [84].

7.3 Hardware Security and Data Protection

FPGA-based systems are typically used in applications and algorithms, like machine learning, that are rapidly changing. The flexibility of FPGA systems is also used by Cloud Service Providers to reuse the resources across multiple users (*tenants*). However, this opens up the possibility that malicious providers can copy the design's intellectual property or users can steal sensitive data of other applications [16]. While protection methods exist for specific cases, their manual application becomes unfeasible for large systems due to their cost or for non-expert designers due to their complexity. Also, the heterogeneity of the system can introduce new vulnerabilities due to the interaction of components that are not designed at the same time. For example, accelerators have fixed functionality, therefore it is not possible to perform *code injection*. However, a malicious attacker can launch **software-based attacks** by providing configuration parameters or system configurations that exploit known vulnerabilities in single components or their interactions.

Physical attacks can exploit the weaknesses of a given hardware implementation for stealing private data (*information leakage*). For example, side-channel attacks can extract secret data by analyzing non-functional effects (like power consumption and timing characteristics) of the device execution. Accelerators can mitigate side-channel attacks by scrambling the execution to make a uniform power consumption or ensuring constant execution time to thwart timing attacks [85]. The modular HLS flow can accommodate additional passes to automatically integrate these extensions and co-optimize them with the accelerator logic.

Another critical issue is **IP theft**: designing heterogeneous systems is a complex and expensive process. The outcome should be protected from reverse engineering and unauthorized copying, which can create billions of dollars of economic damages for the semiconductor design houses. While designers are more sensitive to this problem for ASIC, it is finding an increasing interest also in the FPGA and HLS community. First, designers must guarantee that outsourcing the execution of their designs to third-party cloud providers does not leak details about their algorithms or implementations. Then, embedded FPGAs are also used in several integrated circuits to host specific functions to hide, where HLS identifies and implements such functions to fit into the given logic [86]. Such security features, like watermarking and obfuscation can be introduced on the top of the HLS results [48, 47]. For example, TAO applies semantic obfuscation during HLS to the design of an accelerator that is able to thwart reverse engineering.

8 Conclusion

Thanks to the possibility of reusing logic resources across multiple applications and users, FPGA-based systems are becoming a de facto standard to implement rapidly changing workloads, like modern machine learning applications. High-level synthesis is a key enabling technology for the creation of such systems. Non-expert designers can use HLS to create specialized accelerators directly from high-level specifications, focusing only on the algorithmic development. HLS then creates and optimizes the hardware components based on the user's requirements hiding most of the effort from the designers. This chapter provided an overview on the HLS process, describing the existing approaches for the different HLS phases: the analysis of the input specification, the creation of the accelerator microarchitecture, and the generation of the output files. It also discussed open challenges, like the creation of domain-specific architectures and the programmability issues, and modern challenges, like security concerns, that can be addressed with the support of HLS.

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012.

[2] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[3] Christian Pilato, Stanislav Bohm, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, Vojtech Cima, Radim Cmar, Dionysios Diamantopoulos, Fabrizio Ferrandi, Jan Martinovic, Gianluca Palermo, Michele Paolino, Antonio Parodi, Lorenzo Pittaluga, Daniel Raho, Francesco Regazzoni, Katerina Slaninova, and Christoph Hagleitner. EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE)*, 2021.

[4] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni. Agile soc development with open esp. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2020.

[5] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *Proceedings of the IEEE International SOC Conference*, pages 199–202, 2006.

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[7] Nicola Bombieri, Hung-Yi Liu, Franco Fummi, and Luca P. Carloni. A method to abstract RTL IP blocks into C++ code and enable high-level synthesis. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–9, 2013.

[8] Giovanni De Micheli. High-level synthesis of digital circuits. volume 37 of *Advances in Computers*, pages 207–283. Elsevier, 1993.

[9] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[10] J. Cong. High-level synthesis and beyond - from datacenters to iots. In *Proceedings of the IEEE International System-on-Chip Conference (SOCC)*, pages 1–1, 2015.

[11] A. Cilardo, J. Flich, M. Gagliardi, and R. T. Gavila. Customizable heterogeneous acceleration for tomorrow's high-performance computing. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 1181–1185, 2015.

[12] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, 2015.

[13] S. A. Edwards. The challenges of synthesizing hardware from c-like languages. *IEEE Design & Test*, 23(5):375–386, 2006.

[14] Christian Pilato. Bridging the gap between software and hardware designers using high-level synthesis. In *Proceedings of the International Conference on Parallel Computing (PARCO)*, pages 622–631, 2017.

[15] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[16] C. Pilato, S. Garg, K. Wu, R. Karri, and F. Regazzoni. Securing hardware accelerators: A new challenge for high-level synthesis. *IEEE Embedded Systems Letters*, 10(3):77–80, 2018.

[17] Geoffrey Ndu. *Boosting Single Thread Performance in Mobile Processors using Reconfigurable Acceleration*. PhD thesis, 10 2012.

[18] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[19] Philip Brisk, Adam Kaplan, and Majid Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 395–400, 2004.

[20] Carlo Galuzzi, Elena Moscu Panainte, Yana Yankova, Koen Bertels, and Stamatis Vassiliadis. Automatic selection of application-specific instruction-set extensions. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 160–165, 2006.

[21] N. Pothineni, P. Brisk, P. Ienne, A. Kumar, and K. Paul. A high-level synthesis flow for custom instruction set extensions for application-specific processors. In *Proceedings of the IEEE Asian and South Pacific Design Automation Conference (ASP-DAC)*, pages 707–712, 2010.

[22] L. P. Carloni. From latency-insensitive design to communication-based system-level design. *Proceedings of the IEEE*, 103(11):2133–2151, 2015.

[23] Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. Compositional dataflow circuits. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(1), 2019.

[24] Christian Pilato, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. System-level optimization of accelerator local memory for heterogeneous systems-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):435–448, 2017.

[25] Giuseppe Di Guglielmo, Christian Pilato, and Luca P. Carloni. A design methodology for compositional high-level synthesis of communication-centric socs. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[26] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany. Magnet: A modular accelerator generator for neural networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[27] Jie Liu and Jason Cong. Dataflow systolic array implementations of matrix decomposition using high level synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, page 187, 2019.

[28] Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees Vissers, and Vivek Sarkar. Vyasa: A high-performance vectorizing compiler for tensor convolutions on the xilinx ai engine, 2020.

[29] Jianwen Zhu and Daniel D. Gajski. A unified formal model of ISA and FSMD. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES)*, page 121–125, 1999.

[30] M. Minutoli, V. G. Castellana, A. Tumeo, and F. Ferrandi. Inter-procedural resource sharing in high level synthesis through function proxies. In *Proceedings of the IEEE International Conference on Field programmable Logic and Applications (FPL)*, pages 1–8, 2015.

[31] Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca P. Carloni. ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning. In *Proceedings of the ACM/EDAC/IEEE Design, Automation & Test Conference in Europe (DATE)*, pages 1049–1054, 2020.

[32] Paolo Mantovani, Emilio G. Cota, Christian Pilato, Giuseppe Di Guglielmo, and Luca P. Carloni. Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip. In *Proceedings of the International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 3:1–3:10, 2016.

[33] Christian Pilato, Fabrizio Ferrandi, and Donatella Sciuto. A design methodology to implement memory accesses in high-level synthesis. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 49–58, 2011.

[34] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. Network-attached FPGAs for data center applications. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 36–43, 2016.

[35] Daniel D. Gajski. Silicon compilers and expert systems for vlsi. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 86–87, 1984.

[36] F. Brewer and D. D. Gajski. Chippe: a system for constraint driven behavioral synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(7):681–695, 1990.

[37] K. Bazargan, R. Kastner, S. Ogrenci, and M. Sarrafzadeh. A c to hardware/software compiler. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 331–332, 2000.

[38] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of the International Conference on VLSI Design*, pages 461–466, 2003.

[39] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. *GAUT: A High-Level Synthesis Tool for DSP Applications*, pages 147–169. 2008.

[40] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24:1–24:27, 2013.

[41] Ana Klimovic and Jason H. Anderson. Bitwidth-optimized hardware accelerators with software fallback. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, pages 136–143, 2013.

[42] Stefan Hadjis, Andrew Canis, Ryoya Sobue, Yuko Hara-Azumi, Hiroyuki Tomiyama, and Jason H. Anderson. Profiling-driven multi-cycling in FPGA high-level synthesis. In *Proceedings of the ACM/EDAC/IEEE Design, Automation & Test Conference in Europe (DATE)*, pages 31–36, 2015.

[43] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen D. Brown, and Jason H. Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3):14:1–14:26, 2015.

[44] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Proceedings of the IEEE International Conference on Field programmable Logic and Applications (FPL)*, pages 1–4, 2013.

[45] Marco Lattuada and Fabrizio Ferrandi. A design flow engine for the support of customized dynamic high level synthesis flows. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(4), 2019.

[46] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. TaintHLS: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):798–808, 2019.

[47] Christian Pilato, Kanad Basu, Mohammed Shayan, Francesco Regazzoni, and Ramesh Karri. High-level synthesis of benevolent trojans. In *Proceedings of the ACM/EDAC/IEEE Design, Automation & Test Conference in Europe (DATE)*, pages 1124–1129, 2018.

[48] Christian Pilato, Francesco Regazzoni, Ramesh Karri, and Siddharth Garg. TAO: techniques for algorithm-level obfuscation during high-level synthesis. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–:6, 2018.

[49] Betul Buyukkurt, John Cortes, Jason Villarreal, and Walid A. Najjar. Impact of high-level transformations within the roccc framework. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4), 2011.

[50] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. *Source-to-Source Optimization for HLS*, pages 137–163. 2016.

[51] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 519–530, 2013. ISBN 9781450320146.

[52] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Trans. Archit. Code Optim.*, 14(3), August 2017.

[53] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of Moore's law, 2020.

[54] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. HPVM: Heterogeneous parallel virtual machine. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 68–80, February 2018.

[55] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 296–311, 2018. ISBN 9781450356985.

[56] Marco Lattuada and Fabrizio Ferrandi. Code transformations based on speculative SDC scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 71–77, 2015.

[57] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel & Distributed Systems*, 32(05):1014–1029, 2021.

[58] En-Shou Chang, Daniel D. Gajski, and Sanjiv Narayan. An optimal clock period selection method based on slack minimization criteria. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(3):352–370, 1996.

[59] Leon Stok. Data path synthesis. *Integration*, 18(1):1–71, 1994.

[60] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 433–438, 2006.

[61] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *Proceedings of the IEEE International Conference on Field programmable Logic and Applications (FPL)*, pages 1–8, 2014.

[62] Steve Dai, Gai Liu, and Zhiru Zhang. A scalable approach to exact resource-constrained scheduling based on a joint sdc and sat formulation. In *Proceedings*

of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), pages 137–146.

[63] Christian Pilato, Antonino Tumeo, Gianluca Palermo, Fabrizio Ferrandi, Pier Luca Lanzi, and Donatella Sciuto. Improving evolutionary exploration to area-time optimization of FPGA designs. *Journal of Systems Architecture - Embedded Systems Design*, 54(11):1046–1057, 2008.

[64] P. Ranjan Panda, N. D. Dutt, and A. Nicolau. Incorporating dram access modes into high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(2):96–109, 1998.

[65] C. Pilato, V. G. Castellana, S. Lovergine, and F. Ferrandi. A runtime adaptive controller for supporting hardware components with variable latency. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 153–160, 2011.

[66] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 127–136, 2018.

[67] David C. Ku and Giovanni De Micheli. Constrained resource sharing and conflict resolution in hebe. *Integration*, 12(2):131–165, 1991.

[68] P. Brisk, F. Dabiri, R. Jafari, and M. Sarrafzadeh. Optimal register sharing for high-level synthesis of ssa form programs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(5):772–779, 2006.

[69] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 7–18, 2003.

[70] D. Chen and J. Cong. Register binding and port assignment for multiplexer optimization. In *Proceeding of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 68–73, 2004.

[71] L. Josipovic, P. Brisk, and P. Ienne. From c to elastic circuits. In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers (ACSSC)*, pages 121–125, 2017.

[72] Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, Marco Lattuada, and Fabrizio Ferrandi. Efficient synthesis of graph methods: A dynamically scheduled architecture. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016.

[73] L. Josipovic, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 134–134, 2017.

[74] Jongsok Choi, Stephen D. Brown, and Jason H. Anderson. From pthreads to multicore hardware systems in legup high-level synthesis for fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2867–2880, 2017.

[75] Hsuan Hsiao and Jason H. Anderson. Thread weaving: Static resource scheduling for multithreaded high-level synthesis. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2019.

[76] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanovic. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925*, 2019.

[77] Hosein Mohammadi Makrani, Hossein Sayadi, Tinoosh Mohsenin, Setareh rafatirad, Avesta Sasan, and Houman Homayoun. XPPE: Cross-platform performance estimation of hardware accelerators using machine learning. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2019.

[78] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. Wang. Challenges and trends in modern SoC design verification. *IEEE Design & Test*, 34(5):7–22, 2017.

[79] Pietro Fezzardi, Michele Castellana, and Fabrizio Ferrandi. Trace-based automated logical debugging for high-level synthesis generated circuits. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 251–258, 2015.

[80] Paolo Mantovani, Emilio G. Cota, Kevin Tien, Christian Pilato, Giuseppe Di Guglielmo, Ken Shepard, and Luca P. Carloni. An fpga-based infrastructure for fine-grained dvfs analysis in high-performance embedded systems. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[81] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, June 2020. ISSN 0001-0782.

[82] Fazal Hameed, Asif Ali Khan, and Jeronimo Castrillon. Performance and energy-efficient design of STT-RAM last-level cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(6):1059–1072, 2018.

[83] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[84] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys*, 51(5), 2018.

[85] Zhenghong Jiang, Steve Dai, G. Edward Suh, and Zhiru Zhang. High-level synthesis with timing-sensitive information flow enforcement. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[86] J. Chen, M. Zaman, Y. Makris, R. D. S. Blanton, S. Mitra, and B. C. Schafer. DECOY: DEflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual Property. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.