Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications

Fabrizio Ferrandi*, Vito Giovanni Castellana†, Serena Curzel*, Pietro Fezzardi*, Michele Fiorito*, Marco Lattuada*, Marco Minutoli†, Christian Pilato*, Antonino Tumeo†

*Politecnico di Milano, Italy, †Pacific Northwest National Laboratory, USA

***Abstract*—This paper presents the open-source high-level synthesis (HLS) research framework Bambu. Bambu provides a research environment to experiment with new ideas across HLS, high-level verification and debugging, FPGA/ASIC design, design flow space exploration, and parallel hardware accelerator design. The tool accepts as input standard C/C++ specifications and compiler intermediate representations (IRs) coming from the well-known Clang/LLVM and GCC compilers. The broad spectrum and flexibility of input formats allow the electronic design automation (EDA) research community to explore and integrate new transformations and optimizations. The easily extendable modular framework already includes many optimizations and HLS benchmarks used to evaluate the QoR of the tool against existing approaches [1]. The integration with synthesis and verification backends (commercial and open-source) allows researchers to quickly test any new finding and easily obtain performance and resource usage metrics for a given application. Different FPGA devices are supported from several different vendors: AMD/Xilinx, Intel/Altera, Lattice Semiconductor, and NanoXplore. Finally, integration with the OpenRoad open-source end-to-end silicon compiler perfectly fits with the recent push towards open-source EDA.**

I. The Bambu framework

Bambu is a command-line tool aimed at assisting the designer during the HLS of complex applications. It supports most of the C/C++ constructs, including function calls and sharing of the modules, pointer arithmetic and dynamic resolution of memory accesses, accesses to arrays and structs, parameters passed by reference or copy, and many more. Like in a standard software compilation flow, Bambu has three phases (see Figure 1): front-end, middle-end, and back-end.

Bambu front-end. Bambu interfaces with existing compilers, such as GCC and Clang. With GCC, a plugin extracts the call graph and the control data flow graph of the functions under analysis from GCC's internal IR. Similarly, a Clang plugin extracts the same information and serializes them into a textual format easy to parse. Bambu then parses back all the compiler serialized information plus all the annotations to build a Static Single Assignment in-memory IR. This approach decouples the compiler front-end code from the rest of the HLS process. Localizing all the changes in a GCC or LLVM/Clang plugin allows rapid and easy integration of many different versions of the compilers. Bambu supports GCC versions ranging from 4.5 to 8, and LLVM/CLANG versions ranging from 4.0 to 11. Moreover, the Vivado HLS front-end [2], based on a customized version of LLVM/CLANG and recently released in open-source, was effortlessly integrated into the Bambu framework.

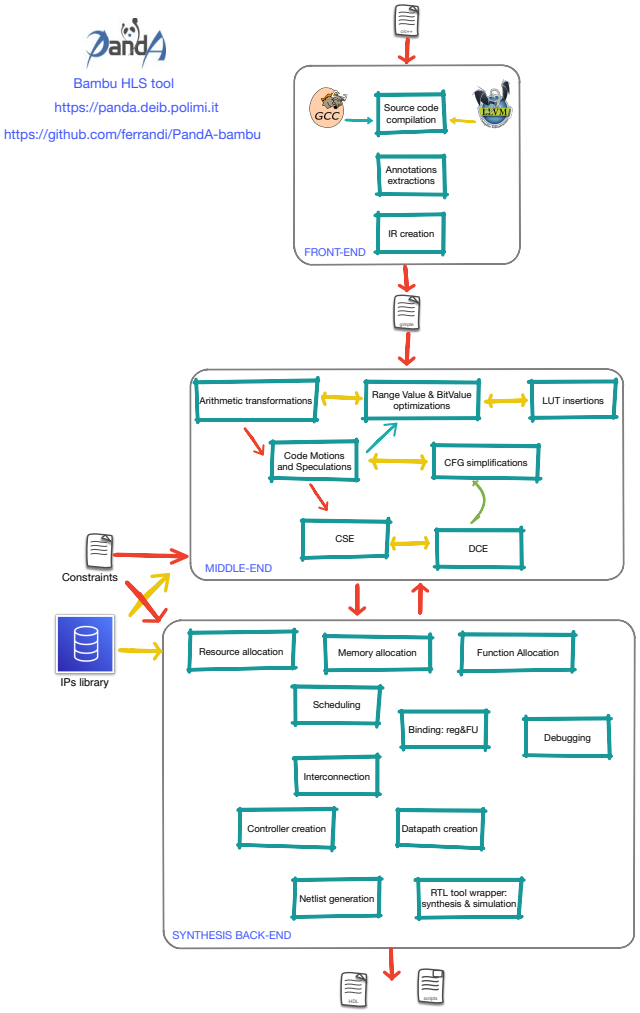

Fig. 1. Bambu Compilation flow.

Bambu middle-end. Starting from the intermediate rep-

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

resentation extracted from GCC/Clang, Bambu rebuilds data structures, such as the Call Graph and the Control Data Flow Graphs, and builds additional data structures such as the Program Dependence Graphs. Next, it applies a set of device-independent analyses and transformations. Some of these steps are commonly used in a software compilation flow (e.g., data flow analysis, loop recognition, dead code elimination, constant propagation, LUT expression insertion, etc.). Multiplications and divisions by constant values are transformed into expressions that use only shifts and adders to reduce area utilization and improve timing. The resulting expression structure depends on the target device and technology, since adders and multipliers may have different performances on different devices. Differently from general-purpose software compilers, designed to target a processor with a fixed-sized data-path (usually 32 or 64 bits), a HLS compiler can exploit custom-size operators (e.g., a multiplier with the minimum number of I/O bits) and registers. Consequently, we can select the minimal number of bits required for the specific algorithm's operations and value storage, which leads to less area, less power, and shorter critical paths. At this stage, Bambu also performs Bitwidth and Range Analysis, aiming at reducing the number of bits required by data-path operators. This analysis is crucial during the optimization process because it impacts all non-functional requirements (e.g., performance, area, power) of a design without affecting its behavior.

Bambu synthesis back-end. In this phase, Bambu performs the actual architectural synthesis of the specification. The synthesis process acts on each function separately. The resulting architecture reflects the structure of the call graph. A single function includes at least two sub-modules: the control logic and the data-path. Control logic modeled as a Finite State Machine handles the routing of the data values and the temporal execution of the operations. The data-path is a custom mux-based architecture with optimized data types to reduce the number of flip-flops and bit-level multiplexers. It implements all the operations and memories required during the function execution. The following paragraphs describe the sequence of steps that Bambu implements to generate control and data-path modules.

Function Allocation. Functions Allocation associates the high-level functions with specific resources available in the technology library associated with the target device. The technology library coming with Bambu integrates standard functions described in Verilog or VHDL, standard system libraries such as `libc` and `libm`, and designer-defined components written in Verilog or VHDL. Bambu supports function pointers and sharing of (sub)modules across module boundaries [3]. Sharing is obtained through function proxies, which act as forwarders of function calls in the original specification to shared modules. Sharing through function proxies provides valuable area savings when complex call graphs are considered, with no significant impact on the execution delays.

Memory Allocation. Memories Allocation defines the memories used to store aggregate variables (arrays and structures), global variables, and how the dynamic memory allocation is implemented. Bambu adopts an architecture for memory accesses that support a wide range of cases. Statically analyzing the memory accesses, Bambu builds a hierarchical data-path where memories can be classified as read-only, local, with aligned or unaligned memory accesses, or which require dynamic resolutions. The memory interconnection accordingly defines multiple busses connecting the load/store components to their respective memories. Dual-port BRAMs or memory controllers with complex parallel channels are supported by replicating such memory interconnections as needed. The same memory infrastructure can also connect to external components (e.g., scratchpads, caches, and DRAMs) or directly to the bus to access off-chip memory. Supporting protocol-based accesses (e.g., FIFO or stream-based access) is obtained by generating specific components that replace the load/store instructions.

Resource Allocation. Resource allocation associates operations not mapped on a function to resource units (RU) available in the resource library. During the middle-end phase, the specification is inspected to identify the characteristics of the operations: these include the type of the operation (e.g., addition, multiplication, etc.) and the types of the operands (e.g., integer, float, etc.). Floating-point operations are supported through the HLS of a soft-float library containing basic soft-float operators, or alternatively by exploiting the FloPoCo software [4], a generator of arithmetic Floating-Point Cores. The allocation step maps operations on the set of available RUs; their characterization includes information such as latency, area, and the number of pipeline stages. Usually, more operation/RU matchings are feasible: in this case, selecting a proper RU is driven by design constraints. The library of RUs used by Bambu is quite rich, and may include several implementations for the same operation. Moreover, the library contains RUs described as templates in a standard hardware description language (i.e., Verilog or VHDL). These templates can be retargeted and customized according to the characteristics of the target technology. In this case, it will be the underlying logic synthesis tool that determines the best architecture to implement each operation (for example, multipliers can be mapped either on dedicated DSP blocks or implemented with LUTs). To perform aggressive optimizations, each library component is annotated with information useful during the entire HLS process, such as resource occupation and latency. Bambu adopts a pre-characterization approach: the performance estimation considers a generic template of the RU, which can be parametric with respect to the bit widths and pipeline stages; latency and resource occupation are then obtained by synthesizing each configuration and storing the resulting metrics in the library as an XML file.

Scheduling. By default, Bambu employs a List scheduling algorithm. In its basic formulation, List scheduling associates each operation with a priority according to particular metrics. The List scheduling proceeds iteratively, associating a set of operations to be executed with each control step. Ready operations (i.e., whose dependencies have been satisfied in previous iterations of the algorithm) can be scheduled in

the current control step considering the availability of the resources. If multiple ready operations compete for a resource, then the one having a higher priority is scheduled. In addition to this old but efficient algorithm, Bambu also features a more aggressive scheduling algorithm, the Speculative scheduling algorithm based on System of Difference Constraints [5]. This algorithm builds an integer linear programming formulation of the scheduling problem, allowing code motion and speculation of operations that belong to different basic blocks.

Module Binding. Within the computed schedule, operations that execute concurrently are not allowed to share the same resource instance. In Bambu, binding is performed through a clique covering algorithm on a weighted compatibility graph [6]. The compatibility graph is built by analyzing the schedule: operations scheduled on different control steps are compatible. Weights express how much it is profitable for two operations to share the same hardware resource. They are computed considering area/delay trade-offs caused by sharing; for example, RUs that occupy a large area will be more likely shared. Weights computation also considers the cost of interconnections required by the steering logic. Bambu also offers several other algorithms for solving the covering problem on compatibility/conflict graphs.

Register Binding. Register binding associates storage values to registers and requires a preliminary analysis step, the liveness analysis [6]. Liveness analysis starts from the schedule to identify each variable's life intervals, i.e., the sequence of control steps in which a temporary value needs to be stored. Variables with non-overlapping life intervals may share the same register.

Interconnection Binding. Interconnections are bound according to the outcome of the previous steps: if a functional or memory resource is shared, then the algorithm introduces steering logic on its inputs. It also identifies the set of control signals that will be driven by the controller.

Netlist Generation. The final architecture is then generated and represented through a hyper-graph, highlighting the interconnection between modules. The netlist generation step translates such representation in a register transfer-level (RTL) description in Verilog or VHDL. The process accesses the resource library, which embeds the RTL implementation of each resource. This process is target-dependent, and the hardware descriptions may differ for different technologies (e.g., ASIC or FPGA) or target devices.

Generation of Synthesis and Simulation Scripts. Bambu automatically generates synthesis and simulation scripts that can be customized via XML configuration files. The RTL-synthesis tools currently supported are AMD/Xilinx ISE, AMD/Xilinx Vivado, Yosis-Vivado, Intel/Altera Quartus, Lattice Diamond, NanoXplore, and OpenRoad. Supported simulators are Mentor Modelsim, Xilinx ISIM, Xilinx XSIM, Verilator, and Verilog Icarus.

II. Research lines

This section summarizes the research topics that we are exploring with Bambu. They range from parallelized hardware accelerator design, dynamic scheduling, verification and debugging, design exploration of the compilation flow, machine learning accelerator design, IR development, and integration with logic synthesis tools.

Parallelization. Current HLS tools generate serial or parallel accelerators for regular, easily partitionable, arithmetic-intensive workloads typical of digital signal processing (e.g., through OpenCL annotations); generally not considering the large data sets and variable latency memory accesses typical of irregular applications and graph algorithms. In Bambu, we introduce an architectural template that exploits both instruction-level and task-level parallelism [7]. The solution includes a hardware scheduler that dynamically binds tasks to the available hardware resources. Bambu introduces single-cycle hardware context switching for components implementing the parallel region to maximize memory parallelism and throughput. Standard OpenMP pragma annotations are parsed by Bambu to extract coarse-grained parallelism. The evaluation shows scalability in area and performance with respect to the solutions based on spatial parallelism with a simple fork-join approach, and even with respect to solution implementing dynamic scheduling. Moreover, the addition of temporal multithreading positively impacts resource consumption.

Dynamic vs static scheduling. Most of the current synthesis methodologies in academic and commercial HLS tools employ the Finite State Machine with data-path model. While this is very successful, more dataflow-oriented approaches are possible. In Bambu, it is possible to control the execution of a task-based parallel application with an adaptive controller [8]. The adaptive controller is composed of a set of interacting control elements that independently manage tasks' execution. These control elements check dependencies and resource constraints at runtime, enabling as-soon-as-possible execution. To support parallel access to shared memories and synchronization, the approach also introduces a novel hierarchical memory interface.

Verification and Debug. Verification and Debug support for circuits generated with HLS has received increasing attention. Current approaches focus on the low-level details of the infrastructure necessary for on-chip debugging. Users need to explicitly instruct the tools about where to place tracepoints and manually inspect the traces to spot malfunctions. As HLS frameworks grow in complexity, this can become a real burden, especially if users have little previous exposure to hardware design and to the HLS internals. Bug identification is very time-consuming and error-prone, especially in complex systems generated with HLS including third-party IPs and hand-written modules. Bambu tackles these problems by introducing a new automated technique for bug identification for HLS-based hardware-generated designs. The methodology compares the HLS-generated hardware execution with the software obtained from the same source code and automatically finds where the discrepancies arise [9].

Design space exploration for HLS Designing, describing, and running a complex customized HLS flow is not a trivial task. State-of-the-art compilers and HLS tools are usually

based on static sequences of passes, possibly declined into different flavors. The sequences can indeed be completely fixed, based on pre-analysis results, or selected after multiple evaluations of different static flows. These approaches have significant limitations since several scenarios cannot be addressed, such as the dynamic update of the set of functions to be processed, complex fixed-point analyses and optimizations composed of sequences of several passes, re-execution of the design flow to exploit the information collected in following stages. The design flow of Bambu follows an approach where the design, the description, and the execution of HLS steps are not precomputed, but rather dynamically built and updated during their execution [10]. This approach allows performing code motions and speculations after the first scheduling step [5]. Another possible place where this dynamic approach may be relevant is in the integration of some preliminary logic synthesis results that could be used to change the scheduling or that may affect LUT expression insertion. Currently, Bambu exploits [11] as fast logic synthesis tool, but nothing prevents to include tools such as the ones used by the OpenRoad design flow [12] or the LSOracle logic synthesis framework [13].

Machine Learning accelerators. HLS has been recently used to speed up the design and development process of FPGA-based accelerators for Machine Learning to address the wide abstraction gap between algorithm development and RTL implementation. For example, both Xilinx's FINN compiler [14] and the hls4ml framework [15] rely on Vivado HLS to generate their designs, adding a pre-processing step where a python-based Machine Learning model is translated into a C++ representation. Bambu can be used within similar design flows: once the input ML model has been translated into either a C/C++ or LLVM representation, it can be processed by Bambu as any other specification. In comparison to other HLS tools, Bambu offers the additional benefits of a broader selection of hardware targets, and the possibility of integrating domain-specific optimizations within the compilation flow described earlier. This is a highly active research topic, considering also that a standard way of describing and compiling ML models does not exist yet. For example, one existing standard format for the exchange of Neural Network models is ONNX: Bambu has already been used to generate hardware accelerators starting from ONNX models, exploiting the TVM compiler [16] to translate them into LLVM code. Alternatively, hls4ml proposes an approach where the pre-processing step relies on a library of C++ components optimized for HLS: the intermediate representation produced by hls4ml is highly specialized and dependent on Vivado HLS, but it will be possible to adapt it so that Bambu can be used instead.

MLIR integration. The introduction of MLIR opened further research lines that may be of interest to Bambu. In the front-end, MLIR dialects that can be translated to an LLVM IR could be used to perform domain-specific optimizations before lowering to LLVM code. For example, exploiting the onnx-mlir project [17] it is possible to progressively translate a Neural Network model into an LLVM IR: analyses and transformations can be added as MLIR passes so that the input to Bambu is optimized for HLS rather than for execution on a CPU/GPU target. Instead, efforts coming from the CIRCT project [18] could be integrated within Bambu itself, to leverage MLIR during the compilation and synthesis flow.

III. Concluding Remarks

This paper presents the open-source HLS research framework Bambu. Bambu supports the synthesis of complex C/C++ specifications and allows the experimentation of new ideas on many problems related to the HLS of complex accelerators. We discussed several possible research lines enabled by Bambu, including parallel hardware accelerator design, dynamic scheduling, verification and debugging, design exploration of complex flows, machine learning accelerator design, IR development, and integration with logic synthesis tools (e.g., OpenRoad).

References

[1] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions CAD Integrated Circuits and Systems*, vol. 35, no. 10, Oct. 2016.

[2] "Xilinx Vitis HLS LLVM 2020.2," https://github.com/Xilinx/HLS.

[3] M. Minutoli *et al.*, "Inter-procedural resource sharing in high level synthesis through function proxies," in *International Conference on Field Programmable Logic and Applications, FPL*, Sept 2015, pp. 1–8.

[4] F. de Dinechin *et al.*, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[5] M. Lattuada and F. Ferrandi, "Code transformations based on speculative SDC scheduling," in *IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '15, Nov 2015, pp. 71–77.

[6] L. Stok, "Data path synthesis," *Integration*, vol. 18, no. 1, pp. 1–71, 1994.

[7] M. Minutoli *et al.*, "Svelto: High-level synthesis of multi-threaded accelerators for graph analytics," *IEEE Transactions on Computers*, no. 01, pp. 1–14, feb 2021.

[8] V. G. Castellana *et al.*, "High-level synthesis of parallel specifications coupling static and dynamic controllers," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Virtual conference, May 17-21, 2021*, 2021, pp. 1–11.

[9] P. Fezzardi and F. Ferrandi, "Automated bug detection for high-level synthesis of multi-threaded irregular applications," *ACM Transactions on Parallel Computing*, vol. 7, no. 4, Sep. 2020.

[10] M. Lattuada and F. Ferrandi, "A design flow engine for the support of customized dynamic high level synthesis flows," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 4, pp. 19:1–19:26, Oct. 2019.

[11] M. Soeken *et al.*, "The EPFL logic synthesis libraries," Nov. 2019, arXiv:1805.05121v2.

[12] T. Ajayi *et al.*, "Toward an open-source digital flow: First learnings from the OpenROAD project," in *IEEE/ACM Design Automation Conference 2019, DAC 2019, June 02-06, 2019*, 2019, p. 76.

[13] W. L. Neto *et al.*, "LSOracle: a logic synthesis framework driven by artificial intelligence: Invited paper," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–6.

[14] M. Blott *et al.*, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[15] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

[16] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[17] T. D. Le *et al.*, "Compiling ONNX neural network models using MLIR," *arXiv preprint arXiv:2008.08272*, 2020.

[18] "CIRCT. Circuit IR compilers and tools," https://github.com/llvm/circt.