In Search of Efficient Hardware Designs: A Multi-Objective Journey through MLIR

Joel A. Quevedo
Doctorado en Ciencias de la Ingeniería
Instituto Tecnológico de Tijuana
Tijuana, México
joel.quevedo201@tectijuana.edu.mx

Yazmin Maldonado
Posgrado en Ciencias de la Ingeniería
Instituto Tecnológico de Tijuana
Tijuana, México
yaz.maldonado@tectijuana.edu.mx

***Abstract*— Field Programmable Gate Arrays are pivotal for digital system implementation, but their inherent complexity has prompted the search for user-friendly design approaches. While Hardware Description Languages like VHDL and Verilog provide higher-level abstractions, they demand substantial hardware expertise and can entail significant design costs. High-Level Synthesis emerges as a promising solution, bridging this gap by translating high-level behavioral descriptions, such as those in C/C++ or Python, into hardware designs. This research proposes a novel approach to High-Level Synthesis design flow with the goal of reducing the designer's exposure to hardware aspects. Taking advantage of a novel compiler infrastructure known as Multi-Level Intermediate Representation our proposed methodology enhances scalability and reusability. In the current stage of our ongoing work, we focus on parameter estimation for area, delay, and power, along with their assignment to the CDFG. Subsequent phases involve the application of Multi-Objective Evolutionary Algorithms to optimize the CDFG, resulting in an optimized graph. Ultimately, the CDFG will be transformed into VHDL code and implemented using popular tools like AMD-Xilinx Vivado, Intel Quartus, or among others. The flexibility of the tool, devoid of technology or vendor specificity, highlights the adaptability and appeal of our approach. This research has the potential to streamline HLS processes, making hardware design more accessible and versatile, thereby benefiting a wide range of applications and technologies.**

Keywords—High-Level Synthesis, MLIR, Multi-Objective Optimization, Field Programmable Gate Array, Hardware Description Languages, Intermediate Representations.

I. INTRODUCTION

The advantages offered by Field *Programmable Gate Arrays* (FPGAs) for implementing digital systems have fueled a search for alternatives that can simplify the design process, making this technology accessible to a broader spectrum of research fields. FPGAs enable the implementation of synchronous designs, ensuring that instructions remain synchronized with the system's clock. Furthermore, these devices possess the capability to perform tasks in parallel, executing processes concurrently. However, all those advantages have a huge limitation; *Hardware Description Languages* (HDLs).

The most commonly used HDLs include VHDL and Verilog. These languages enable users to design and debug digital systems at a higher level of abstraction than describing the entire circuit manually on schematic. However, designing a high-quality digital circuit requires a deep understanding of hardware design, alongside careful consideration of numerous factors in the design exploration process, which can potentially become cost-prohibitive [1]. An attractive solution to this problem is *High-Level Synthesis* (HLS) which facilitates the generation of hardware designs from behavioral descriptions written in high-level languages like C/C++ or Python. This methodology offers many advantages in the hardware design process, including reduced development time and more convenient methods for verification and debugging. Additionally, it extends an invitation to software designers to explore the field of hardware design. However, HLS has limitations. Designers need to consider hardware-specific factors, such as timing and resources, to get the best results. Moreover, this design process often does not produce implementations as efficient as those created manually [2][3].

Software compilation and HLS both aim for similar optimization goals, such as removing unnecessary code, simplifying constants, and optimizing loops, just to name a few. Because of this, HLS tools often use structures from software compilation to make these improvements. A widely used tool for this purpose is the *Low-Level Virtual Machine* (LLVM) project [4], which serves as an open-source compiler infrastructure. Most HLS tools use LLVM as a front-end to generate code that works independently across different targets [5].

Despite its popularity, LLVM IR (Intermediate Representation) has a limitation, it operates at a single abstraction level to interface with the system. This limitation becomes apparent when dealing with specific domain-related problems, as many of these problems are better addressed using different levels of abstraction. To overcome this limitation, several programming languages such as Swift, Rust, and Julia have developed their own IRs to focus on domain-specific problems and improve the implementation process. In response to these challenges in programing language design and implementation, is proposed the *Multi-Level Intermediate Representation* (MLIR) [6]. MLIR is a compilation framework that supports multiple levels of functional and representational hierarchy. Its ability to model various levels of abstraction provides compilers with a pathway to create domain-specific IRs.

Currently, there are many HLS tools available, both commercial and academic, such as, Vitis HLS [7], Intel HLS [8] and BAMBU [9]. However, to generate a high-efficient digital circuit, the user still needing a deep knowledge of hardware to indicate the tool through directives the optimizations they need to achieve their design goal [2]. The HLS process often operates within a framework called *Design Space Exploration* (DSE). This framework assists in selecting the best design that aligns with project goals by employing techniques such as scheduling, allocation, and binding. It is important to note that these optimization techniques inherently involve multiple objectives, with several functions to be improved simultaneously [10][11]. The most common objectives that these tools seek to minimize are area, delay, and power consumption.

In the following sections, we will provide detailed information about the current state of HLS tools, with a focus on features relevant to our work, including input/output languages and IR.

The limitations involving the HLS process, including the requirement for solid hardware expertise, the diverse approaches to solving the same problem, and the necessity to perform multi-objective optimizations, have motivated the exploration of alternatives methods to enrich this process. To mitigate these limitations, we propose the utilization of *Multi-Objective Evolutionary Algorithms* (MOEAs) to optimize a CDFG extracted from the MLIR compiler infrastructure. Furthermore, extracting specific information from MLIR's dialects contributes to enhancing the quality of the CDFG.

This study aims to introduce a novel approach to tackle some of the existing challenges in the field of HLS. We propose an end-to-end design flow, starting from a high-level programming language, to HDL code. The HDL code as output will be optimized by the MOEA during the process, considering three objective functions (delay, area, and power).

This paper is organized as follows: Section II presents a theoretical background related to High-Level Synthesis, LLVM/MLIR and the role of Multi-Objective Evolutionary Algorithms in this process. Section III presents a general overview of the current approaches and design methodologies presented in the literature. Section IV presents the details of the proposed methodology and the current stage of the project. Finally, Section V presents the conclusion and future work.

II. Theoretical Background

This section aims to provide a theoretical background related to the process of High-Level Synthesis, including the infrastructure commonly used in these tools. Furthermore, it offers a comprehensive overview of the MLIR project and Multi-Objective Evolutionary Algorithms, both of which play key roles in the context of this work.

A. High-Level Synthesis

In short terms, HLS offers the capability to translate a behavioral description, typically written in a high-level programming language like C/C++, SystemC, or Python, into a hardware description that retains equivalent semantics. A general overview of the HLS design process is shown in Fig. 1, the initial step involves the compilation of the high-level language description into an *intermediate representation*, which encapsulates critical information regarding data dependencies and control flow relationships among operations. Many HLS tools leverage different techniques for representing and analyzing the input code, such as utilizing the *abstract syntax tree* (AST) of C/C++ code, employing a traditional software compiler IR, or constructing a *control/data flowgraph* (CDFG) [12][13]. Subsequently, the IR undergoes a series of critical processes: scheduling, allocation, and binding.

- Scheduling: Determines how individual operations are scheduled across clock cycles, establishing the temporal order in which operations will be executed.
- Allocation: Selects the number and type of hardware resources, including *Functional Units* (FU), connectivity components, and storage elements, necessary to meet design constraints. These components are typically chosen from a hardware resource library that includes characteristics such as area, delay, and power consumption.
- Binding: Dictates how each variable is linked to a specific FU in each clock cycle.

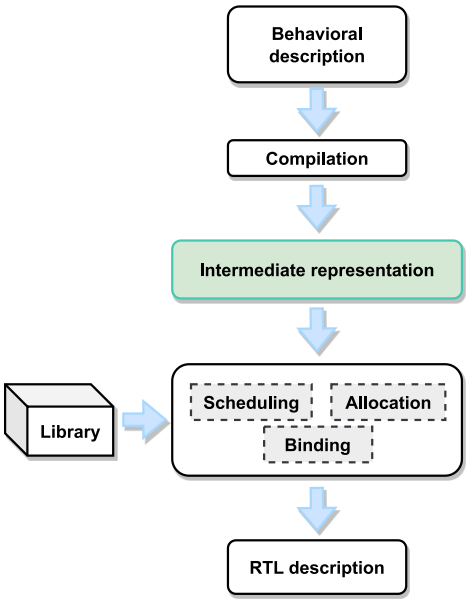

Fig. 1. General diagram of the HLS process.

The DSE framework helps the HLS tool identify the optimal design by exploring various combinations of scheduling, allocation, and binding decisions. Finally, considering the outcomes of these preceding tasks, is generated a *Register-Transfer Level* (RTL) description [11][12]. This RTL description serves as a lower-level representation of the hardware design, ready for further synthesis and implementation. Notice that the IR section is highlighted (Fig. 1), as its structure has a direct and significant impact on the capabilities of the HLS tool. In essence, the more robust and powerful the IR structure, the more proficient the HLS tool becomes in effectively translating the input code and executing optimizations [14].

B. Control and Data Flow Graph

Data dependencies can be effectively illustrated using a *data flow graph* (DFG) where each node corresponds to an operation, and the connections between nodes represent input, output, and temporary variables. However, this structure models only data dependencies and no the control dependencies. This limits the use of DFG representation to a few applications. It is possible to extend the traditional DFG model by adding control dependencies and giving as result a *control and data flow graph* (CDFG). A CDFG is a directed graph in which the edges represent the control flow. The nodes in a CDFG are commonly referred to as *basic blocks* (BB). A BB is itself a DFG, a sequence of statements that contain no branches or internal entrance or exit points. A CDFG exhibits data dependencies inside BBs and captures the control flow between them [12][15]. As an example, in Fig. 2, is presented the CDFG of an *if-else* sentence. At first, the code will evaluate the condition (green node), considered as an entry point, if the condition is *true* the code will add 20 to a variable, if the condition is *false*, will add the variable *b* instead (blue nodes). At the end, it returns the result *x* value (yellow node).

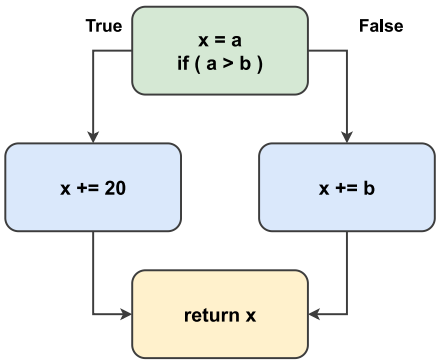

Fig. 2. Example of a CDFG for an *if-else* sentence.

C. LLVM/MLIR Compiler infrastructure

As the MLIR project was inspired by LLVM, we consider that a brief overview of how LLVM works, will help the reader to visualize the advantages.

LLVM [4] is a constantly growing project, that has been widely used for implementation of programming languages, and has become an essential resource for compiler research. In Fig. 3, is shown a basic example of how a C function can be transformed into a LLVM IR. This IR is composed by a *control-flow graph* (CFG) of labeled basic blocks and branches as CFG edges. It also contains the so called φ-functions which purpose is to select the control-flow dependent values, defined in predecessor basic blocks. It's important to note that within each basic block, there exists an internally constructed DFG including the operations and values defined and referenced. In order to illustrate the relationship between the CDFG (Fig. 2) and the LLVM IR, the color of the nodes corresponds to the sections marked by colored lines.

LLVM IR is a *static single assignment* (SSA) IR, which means that each value, known as SSA operand, is assigned exactly once. It is important to notice that, during the lowering from a high-level AST to LLVM IR, certain information from the source code will inevitably be lost [1].

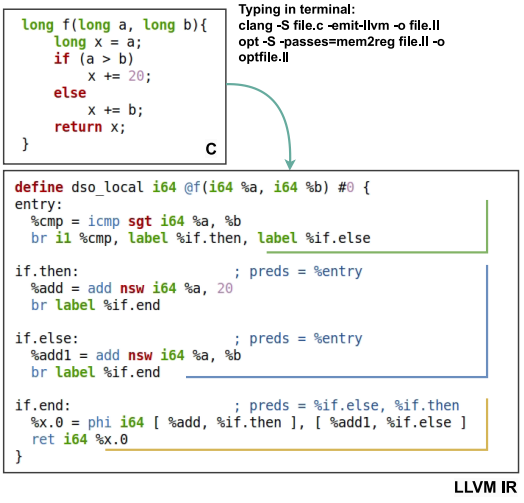

Fig. 3. Example of a long function lowered to LLVM IR through Clang.

Similar to LLVM, MLIR [6] includes a SSA style IR. This infrastructure, allows the user to capture domain-specific IRs into a *Dialect*. These dialects are collection of operations, attributes, and types that describe a particular domain. MLIR offers a versatile platform with both predefined dialects for common functionalities and an open framework that empowers users to create custom dialects. The shared semantics across all dialects are defined by the infrastructure itself, utilizes SSA values. In MLIR a sequential list of operations without control flow is defined as a BB, and a CFG of blocks is structured within a *Region*. Moreover, operations have the capability to include regions, enabling the representation of arbitrary design hierarchies.

In Fig. 4, we observe two different representations of the same *f* function within MLIR. Initially, the function is translated into SCF dialect. In this representation, the structure defines the constant (`%c20_i64`), followed by the computation of the as a Boolean value (`%0`). Then, it enters to `scf.if` operation, where the addition of the arguments is evaluated accordingly. Next, one can further lower the SCF dialect to Standard dialect, with blocks being used the same as BB in LLVM IR, where branches (`br`) are used for control flow. As same as previous figures, the colored lines correspond to the color in CDFG nodes.

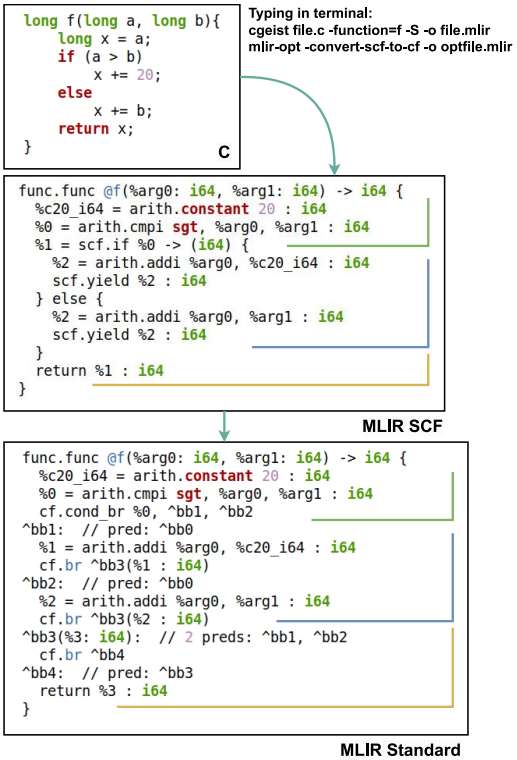

Fig. 4. Example of a long function lowered to MLIR Standard through Polygeist [39] and MLIR.

D. Multi-Objective Optimization in HLS

In HLS the DSE can be classified as a multi-objective optimization problem, since the main goal is to minimize a set of conflicting design parameters [16]. As mentioned earlier, there are many possible optimizations during the process scheduling, allocation, and binding. These optimizations involve conflicting objective functions, by this reason, the

problem typically needs the application of a multi-objective optimization algorithms (MOAs). MOAs maintain a trade-off between conflicting metrics. The literature has reported the optimization of many objective functions during the HLS process, such as, delay, area, power, wirelength, digital noise, reliability, temperature, and security [11]. This work is mainly interested in the simultaneous optimization of:

- Delay: Also called, timing, latency, or performance correspond to the total number of time steps or clock cycles.
- Area: Total number of hardware resources necessary to meet design constraints.
- Power: Total power consumption.

All of these objective functions, are expected to be minimized by the algorithm.

Multi-Objective Evolutionary Algorithms (MOEAs) have proven to be valuable tools for the DSE in HLS, helping the designers to select the design that aligns the best with project requirements. In the literature, many authors explore the integration of MOEAs in the HLS process [10][17][18]. However, in [19], is conducted a comparative analysis using quality indicators to evaluate the performance of some algorithms, including NSGA-II (Nondominated Sorting Genetic Algorithm II) [20], NSGA-III (Nondominated Sorting Genetic Algorithm III) [21] and SPEA2 (Strength Pareto Evolutionary Algorithm 2) [22]. These studies collectively demonstrate the feasibility and effectiveness of implementing MOEAs in the HLS process.

III. Related Work

In the literature, various approaches and design methodologies have been documented, all sharing a common goal: addressing the limitations inherent in current HLS processes.

Over the years, many HLS tools have been implemented [23]. Each of them offering distinct features, including variations in input/output languages, intermediate representations, their own internal optimizations, or specialization for specific devices or technologies. Also, these tools can be commercial or academic. Their general information is listed in Table I and Table II accordingly.

TABLE I. HLS ACADEMIC TOOLS.

Tool Name	*Input Language*	*Output Language*	*IR*
LegUp [24]	C	Verilog	DFG, CDFG, LLVM
BAMBU [9]	C	Verilog/VHDL	CDFG, Call Graph, LLVM
Dynamatic [25]	C/C++	VHDL	CDFG, LLVM
DWARV [26]	Subset C	VHDL	CDFG
GAUT [27]	C/C++	VHDL	CDFG

TABLE II. HLS COMERCIAL TOOLS.

Tool Name	*Input Language*	*Output Language*
Vitis HLS[a] [7]	C/C++, SystemC	Verilog, VHDL
Intel HLS [8]	C/C++	Verilog, VHDL, System Verilog
Smart HLS [28]	C/C++	Verilog
CyberWorkBench [29]	BDL	Verilog, VHDL
Catapult-C [30]	C/C++, SystemC	Verilog, VHDL
Stratus [31]	C/C++, SystemC	RTL
Bluespec [32]	BSV	Verilog
Synphony [33]	M-language	RTL, C-model
MaxCompiler [34]	MaxJ	RTL

[a.] Uses LLVM as an IR

As shown in the tables, most of the tools primarily use C/C++ languages to describe the input program. Additionally, Verilog and VHDL are equally popular choices. Academic tools have limited information available, creating a barrier to identify their components, such as, the IR and its properties. However, the HLS community has introduced innovative design flows that incorporate high-level languages like Python [35][36] and Julia [37]. Some alternative approaches leverage the MLIR infrastructure to facilitate the transformation of code into an HDL [1][14][38].

Despite the variety of design flows, designers are still required to address hardware-specific considerations. Furthermore, for our knowledge base, none of these approaches prioritize the concurrent optimization of area, latency, and power.

IV. Proposed Methodology

In order to contribute with an alternative for HLS process, we propose a design flow, which includes in its middle-end a MLIR and a CDFG that will be optimized by a selected MOEA. This aims to mitigate the limitations addressed in the Fig. 5, and reduce the hardware knowledge needed to achieve an optimal solution of a digital circuit. Also, the utilization of a novel compiler infrastructure like MLIR, offers an interesting entry point to explore and research their effect in the application of multi-objective optimization in HLS.

A specific diagram of the proposed design flow is presented in Fig. 6. As a front-end, Polygeist [39] is a suitable option to transform C/C++ input code to MLIR. At this step, we propose to extract a CDFG from MLIR which nodes must contain the information related to delay, area, and power objective functions. These parameters, can be estimated as described in [40][41]. Once the CDFG is constructed, a multi-objective optimization will be performed by the selected MOEA (SPEA2, NSGA-II, NSGA-III). Finally, the optimized

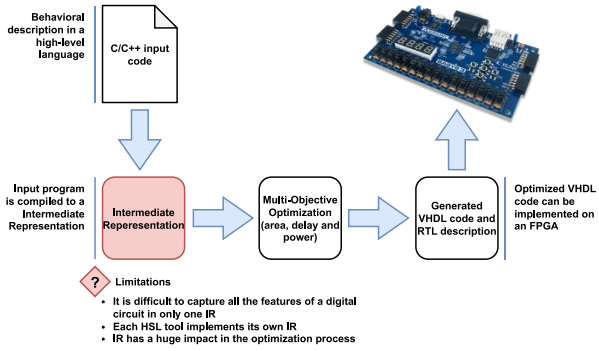

Fig. 5. Illustrative diagram addressing the problem under study.

CDFG will be translated to a VHDL code with equivalent semantics.

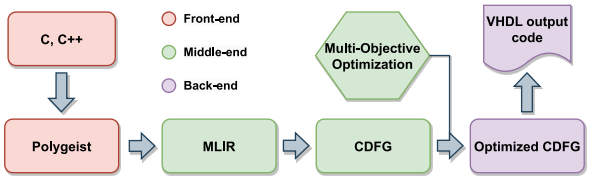

Fig. 6. Diagram of the proposed design flow.

At this stage of the project, we are working on the construction of the CDFG with the attributes corresponding to the objective functions to minimize (area, delay and power). Further, we will perform experiments applying the MOEAs to the CDFG mentioned before. Once this step is validated, we will continue with the translation of the CDFG into the VHDL code. The generated VHDL code must be ready to synthesize and implement into an FPGA using a platform like Xilinx Vivado [7] or Intel Quartus[8].

We are particularly interested in a software tool called "VHDL by MOEA" [42], which focuses on performing multi-objective optimizations using a DFG extracted from input C code. This tool offers several advantages, including the visualization of the IR and the ability for users to select a MOEA along with its parameters, such as population size, generations, crossover, and mutation probability. Also, it produces optimized VHDL code as its output. However, "VHDL by MOEA" currently has limitations as it utilizes a DFG as its IR, which restricts the allowed C instructions. To address this limitation and enable more complex designs, we propose upgrading the tool to support a CDFG by leveraging the benefits of MLIR. This enhancement will make the tool more robust and versatile, allowing it to handle a wider variety of C code.

We believe that integrating our proposed design flow with "VHDL by MOEA" presents an immediate and highly advantageous application of our research. This integration will significantly augment the tool's capabilities, opening the door to a more extensive range of code compatibility, aligning well with the goals of our research.

V. Conclusion

In this paper, we have presented a novel HLS design flow proposal aimed at reducing the designer's exposure to hardware aspects, effectively delegating these responsibilities to the optimization algorithm. The integration of MLIR compiler infrastructure adds significant flexibility to explore various design flows. For instance, it allows the implementation of a front-end that can translate from other high-level languages, such as Python (see nelli [43]), to an MLIR dialect, thus providing access to the methodology proposed in this work. Additionally, the back-end can be modified to convert the optimized CDFG into HDLs like Verilog or SystemVerilog, offering versatility in implementing digital circuits.

As previously mentioned, this work is still in progress. The current stage involves parameter estimation for area, delay, and power, along with their assignment to the CDFG. Following this, we will proceed with the application of MOEAs to optimize the CDFG, resulting in an optimized graph. Ultimately, the CDFG will be translated into VHDL code and implemented using external tools such as Xilinx Vivado or Intel Quartus. The ability to select the implementation tool, which is not tied to any specific technology, vendor, or device, highlights the attractiveness of our proposed approach.

HLS has proven successful in domains such as deep learning, video transcoding, graph processing, and bioinformatics [44]. These are areas where our proposed design flow can provide an interesting solution for hardware implementation. Moreover, we expect that the methodology presented in this work serve as a base for future research in the HLS flow and encourage the community to explore new optimization techniques or the integration and evaluation of other multi-objective evolutionary algorithms.

This ongoing research holds the potential to significantly enhance the HLS process and reduce the complexity traditionally associated with hardware design, making it more accessible and versatile for a broader range of applications and technologies.

Acknowledgment

This work was supported by CONAHCYT scholarship 1014860, and the Tecnológico Nacional de México with de project 17615.23-P.

References

[1] Morten Borup Petersen. "A Dynamically Scheduled HLS Flow in MLIR". Masters Thesis. École polytechnique fédérale de Lausanne, 2022.

[2] L. Huang, D.-L. Li, K.-P. Wang, T. Gao, and A. Tavares, "A Survey on Performance Optimization of High-Level Synthesis Tools," *J. Comput. Sci. Technol.*, vol. 35, no. 3, pp. 697–720, May 2020, doi: 10.1007/s11390-020-9414-8.

[3] A. S. Canas Ferreira, "Restructuring Software Code for High-Level Synthesis Using a Graph-based Approach Targeting FPGAs," Universidade Do Porto, 2018.

[4] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86. doi: 10.1109/CGO.2004.1281665.

[5] S. Ravi and M. Joseph, "Open source HLS tools: A stepping stone for modern electronic CAD," in *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, Chennai: IEEE, Dec. 2016, pp. 1–8. doi: 10.1109/ICCIC.2016.7919615.

[6] C. Lattner *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Seoul, Korea (South): IEEE, Feb. 2021, pp. 2–14. doi: 10.1109/CGO51591.2021.9370308.

[7] AMD/Xilinx. *Vitis HLS*. Available online: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls.

[8] Intel. *Intel HLS Compiler*. Available online: https://www.intel.com/content/www/us/en/docs/programmable/683456/21-4/pro-edition-user-guide.html.

[9] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sep. 2013, pp. 1–4. doi: 10.1109/FPL.2013.6645550.

[10] M. C. Bhuvaneswari, Application of evolutionary algorithms for multi-objective optimization in VLSI and embedded systems. Springer, 2014.

[11] D. Reyes Fernandez de Bulnes, Y. Maldonado, and L. Trujillo, "Development of Multiobjective High-Level Synthesis for FPGAs," *Scientific Programming*, vol. 2020, pp. 1–25, Jun. 2020, doi: 10.1155/2020/7095048.

[12] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 8–17, Jul. 2009, doi: 10.1109/MDT.2009.69.

[13] H. Ye *et al.*, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation." arXiv, Dec. 22, 2021. Accessed: Sep. 21, 2023. [Online]. Available: http://arxiv.org/abs/2107.11673

[14] M. Urbach and M. B. Petersen, "HLS from PyTorch to System Verilog with MLIR and CIRCT". *LATTE'22*, March. 2022.

[15] J. Cheng, L. Josipovic, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining Dynamic & Static Scheduling in High-level Synthesis," pp. 288–298, Feb. 2020, doi: 10.1145/3373087.3375297.

[16] B. C. Schafer and Z. Wang, "High-Level Synthesis Design Space Exploration: Past, Present, and Future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2628–2639, Oct. 2020, doi: 10.1109/TCAD.2019.2943570.

[17] A. Sengupta and R. Sedaghat, "Integrated scheduling, allocation and binding in High Level Synthesis using multi structure genetic algorithm based design space exploration," in *2011 12th International Symposium on Quality Electronic Design*, Mar. 2011, pp. 1–9. doi: 10.1109/ISQED.2011.5770772.

[18] B. C. Schafer and K. Wakabayashi, "Machine learning predictive modelling high-level synthesis design space exploration," *IET Computers & Digital Techniques*, vol. 6, no. 3, pp. 153–159, May 2012, doi: 10.1049/iet-cdt.2011.0115.

[19] D. R. Fernández de Bulnes and Y. Maldonado, "Comparación de Algoritmos Evolutivos Multi-Objetivo para Síntesis de Alto Nivel en dispositivos FPGA," *CyS*, vol. 22, no. 2, Jul. 2018, doi: 10.13053/cys-22-2-2946.

[20] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II," in *Parallel Problem Solving from Nature PPSN VI*, M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 849–858. doi: 10.1007/3-540-45356-3_83.

[21] H. Jain and K. Deb, "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 602–622, Aug. 2014, doi: 10.1109/TEVC.2013.2281534.

[22] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm," 2001. doi: 10.3929/ETHZ-A-004284029.

[23] R. Nane *et al.*, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016, doi: 10.1109/TCAD.2015.2513673.

[24] Andrew Canis et al. "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.2 (2013), pp. 1–27.

[25] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. "Invited tutorial: Dynamatic: From C/C++ to dynamically scheduled circuits". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 1–10.

[26] Razvan Nane et al. "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler". In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2012, pp. 619–622.

[27] Philippe Coussy et al. "GAUT: A high-level synthesis tool for DSP applications". In: *High-Level Synthesis*. Springer, 2008, pp. 147–169.

[28] Microchip. *Smart HLS*. Available online: https://microchiptech.github.io/fpga-hls-docs/.

[29] NEC. *CyberWorkBench*. Available online: https://www.nec.com/en/global/prod/cwb/pdf/CWB_Detailed_technical.pdf.

[30] Siemens. *Catapult C++/SystemC Synthesis*. Available online: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/.

[31] Cadence. *Stratus HLS*. Available online: https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[32] Bluespec. *Bluespec Compiler*. Disponible en: https://bluespec.com/.

[33] Synopsis. *Synphony HLS*. Available online: https://news.synopsys.com/index.php?s=20295&item=123096.

[34] Maxeler. *MaxCompiler*. Available online: https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf.

[35] MyVHDL. Available online: https://www.myhdl.org/.

[36] PyVHDL. Available online: https://pyvhdl-docs.readthedocs.io/.

[37] B. Biggs, I. McInerney, E. C. Kerrigan, and G. A. Constantinides, "High-level Synthesis using the Julia Language." arXiv, Feb. 17, 2022. Accessed: Sep. 21, 2023. [Online]. Available: http://arxiv.org/abs/2201.11522

[38] CIRCT. [n.d.]. Circuit IR Compilers and Tools. Online. https://circt.llvm.org

[39] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising C to Polyhedral MLIR," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2021, pp. 45–59. doi: 10.1109/PACT52795.2021.00011.

[40] D. S. H. Ram, M. C. Bhuvaneswari, and S. M. Logesh, "A Novel Evolutionary Technique for Multi-objective Power, Area and Delay Optimization in High Level Synthesis of Datapaths," in *2011 IEEE Computer Society Annual Symposium on VLSI*, Jul. 2011, pp. 290–295. doi: 10.1109/ISVLSI.2011.55.

[41] E. Kursun, R. Mukherjee, and S. O. Memik, "Early Quality Assessment for Low Power Behavioral Synthesis," *Journal of Low Power Electronics*, vol. 1, no. 3, pp. 273–285, Dec. 2005, doi: 10.1166/jolpe.2005.028.

[42] Darian Reyes Fernández de Bulnes. "Optimización de recursos para plataformas reconfigurables mediante metaheurísticas". PhD thesis. Instituto Tecnológico de Tijuana, 2019. VHDL by MOEA available: http://201.174.122.25/vhdlbymoea/

[43] M. Levental, A. Kamatar, R. Chard, K. Chard, and I. Foster, "nelli: a lightweight frontend for MLIR." arXiv, Aug. 14, 2023. Accessed: Sep. 22, 2023. [Online]. Available: http://arxiv.org/abs/2307.16080

[44] J. Cong et al., "FPGA HLS Today: Successes, Challenges, and Opportunities," ACM Trans. Reconfigurable Technol. Syst., vol. 15, no. 4, p. 51:1-51:42, Aug. 2022, doi: 10.1145/3530775.