Is It a Good Idea to Build an HLS Tool on Top of MLIR? Experience from Building the Dynamatic HLS Compiler

Jiahui Xu
ETH Zurich
Zurich, Switzerland
jxu@ethz.ch

Emmet Murphy
ETH Zurich
Zurich, Switzerland
emurphy@ethz.ch

Lana Josipović
ETH Zurich
Zurich, Switzerland
ljosipovic@ethz.ch

Abstract

When the MLIR project was first introduced, it promised to address the issues that the HLS community had with the LLVM project. But is this really the case, and is MLIR the "right"/"best" compiler infrastructure for HLS? We here share our experiences based on the development of Dynamatic (github.com/EPFL-LAP/dynamatic).

1 Introduction

The LLVM project [1] has been a foundation for many open-sourced and commercial *high-level synthesis* (HLS) projects [2–7], that compile high-level software code to RTL. However, LLVM is not ideal for HLS tools [8]: the IR cannot be customized to represent circuits, which forces HLS tools to create custom and hard-to-reuse IRs for circuits.

MLIR [8, 9] promises to solve this issue: it introduces a standard way to define, create, analyze, and transform custom IR operations. With MLIR, HLS developers can overcome the rigidity of LLVM IR: they can define high-level IRs that benefit from high-level transformations or low-level IRs with circuit semantics. For custom IRs, MLIR provides some default C++ routines for creating and manipulating IR objects, dumping and parsing their textual format, and so on. This greatly reduces implementation overhead and encourages interoperability between the MLIR-based tools.

As some of the main developers of Dynamatic [4, 5]—an MLIR-based HLS tool—we acknowledge and leverage its abovementioned benefits. Yet, we have also recognized some features of it that create obstacles. This paper shares with the LATTE community the issues we encountered with MLIR (as of version d8eb4ac): the IR definition, analysis, and transformation, and integration between different MLIR-based HLS projects. We believe that our observations are general and go beyond HLS MLIR projects. We hope that these findings will bring new insights and discussions in the MLIR community and help the developers to make better decisions for future HLS tools.

2 Background

This section discusses the relevant background of MLIR and Dynamatic.

MLIR [9] is a compiler infrastructure that helps define custom IR (called dialects) and transformation passes. The IR contains *operations*, which consume and produce *values*.

Dynamatic produces *dataflow circuits* [10, 11], which consist of *dataflow units* of instruction granularity connected via *handshake channels*; the data is encapsulated in a token, exchanged via handshake channels. In dataflow circuits, operations execute whenever their inputs are valid. Therefore, Dynamatic produces dynamically-scheduled circuits that have a performance advantage whenever the control flow or memory access pattern is unpredictable [12–14]. Dynamatic is an MLIR-based compiler: it represents units and channels as operations and values in the specialized *handshake* dialect.

Dynamatic has evolved beyond a research prototype: It forms the basis for numerous publications [11–37]—many of which have been incorporated into the main HLS flow [11–13, 15, 16, 21–25, 28, 32]. We have held multiple tutorials and talks in technical conferences [5, 38]. Dynamatic merges approximately 30 pull requests (PR) every month into its main branch, and every PR is monitored with an automated CI/CD pipeline to prevent compilation errors or performance regression. Code contributions from less experienced developers will receive detailed reviews and feedback.

3 Building an MLIR-Based HLS Compiler in an Academic Setting

We acknowledge the significant advantages of using MLIR as the basis for an HLS project in an academic setting. Dynamatic is mainly built by student developers. Many of us have an electrical engineering background, and we are not professionally trained software developers; often, we are unaware of best software practices. MLIR helps us overcome this challenge: it provides insight into how a complex object-oriented programming architecture works in practice, and its organization has inspired us to create a modular, easy-to-understand tool. These benefits enable us to dedicate more development effort to HLS-specific features.

However, we also recognize that MLIR has features that create obstacles for HLS tools, as we discuss next.

4 Limitations of MLIR for Open-Source HLS Projects and Case Studies in Dynamatic

This section discusses the issues that we encountered when building the Dynamatic HLS tool on top of MLIR.

4.1 MLIR Value is Not a Fit for Modeling Graph Edges

Modeling software IRs or circuits as graphs is a common practice [39]. Some information naturally belongs to the nodes, and some belongs to the edges. MLIR allows annotations (called *attributes*) on operations. However, no attribute can be attached to MLIR values, representing data exchanged between operations. How does an MLIR HLS framework annotate edge information?

Case study 1: memory dependencies. Figure 1 describes a C program and a fraction of the corresponding MLIR. There is a potential read-after-write dependency between `store3` and `load2` from the following iteration. This dependency is typically determined by software IR analysis; a *dependency edge* between the store and the load is annotated with the *dependence distance* and used as an HLS scheduling constraint to guarantee correct operation order. As MLIR disallows edges or operation pairs to be annotated, this information must be represented in a non-standard and non-intuitive way.

Dynamatic uses alias and polyhedral analysis to determine the dependency pairs between memory accesses [13, 15]. To maintain this information in the HLS flow, Dynamatic assigns every operation a unique name, maintains the name throughout the HLS pipeline, and uses a hard-to-read format to represent the dependency. For correctness, Dynamatic must additionally ensure the validity of the dependency by guaranteeing that every transformation preserves the pairwise validity of the names.

Case study 2: recording software profiling. Software profiling is typically used in HLS to determine operation execution counts and sequences (e.g., for early performance estimates or targeted optimizations of frequently executed constructs). Although this information is naturally attributed to control-flow edges between basic blocks, there is no standard way to annotate control-flow edges in MLIR.

Dynamatic relies on software profiling to identify the frequently executed loops that should be prioritized in optimization [16, 22]. This information could have been recorded on the output edges of the branch units (see the `cond_br` nodes in Figure 2b). Yet, since MLIR does not allow value annotation, Dynamatic had to rely on an external CSV file to hold this information.

Proper edge-annotation support in MLIR would alleviate these issues.

4.2 Are Block Arguments Convenient for HLS Conversion?

In an HLS flow, we need to convert a software representation into a circuit. In particular, we have to convert SSA ϕ nodes (representing a conditional assignment) into multiplexers. Is this conversion straightforward in MLIR?

```
void histogram(in_int_t feature[1000], in_float_t weight[1000],
               inout_float_t hist[1000], in_int_t n) {
  for (int i = 0; i < n; ++i) {
    int m = feature[i];
    float wt = weight[i];
    float x = hist[m]; // -> "name = load2"
    hist[m] = x + wt; // -> "name = store3"
  }
}
```

(a) A program with a *read-after-write* (RAW) dependency.

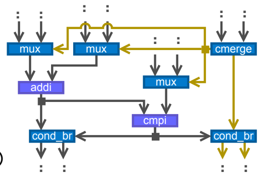
```
%9 = memref.load %feature[%8] {
  handshake.name = "load2"
  } : memref<1000xf32>
...
// The dependency information, which conceptually
// is an edge "store3 -> load2" is annotated
// on the node in an awkward way.
memref.store %10, %feature[%11] {
  handshake.deps = #handshake<deps[["load2", 1]]>,
  handshake.name = "store3"
  } : memref<1000xf32>
```

(b) The produced MLIR snippet.

Figure 1. Annotating edge information in MLIR is tricky.

```
^bb1(%3 : i32, %4 : i32):
%5 = addi %3, %2 : i32
%pred = cmpi slt %4, %5 : i1
cf.cond_br %pred,^bb2(%5),^bb3(%5)
```

(a)



(b)

Figure 2. Dynamatic aims to translate software IR (left) to dataflow circuit (right).

Case study 3: convert block arguments to multiplexers. Figure 2a describes a basic block in the ControlFlow dialect (a built-in MLIR dialect for representing sequential programs); it begins with an identifier (bb1) with the *block arguments* (`%3, %4`) and ends with a conditional branch. MLIR represents the ϕ nodes with these block arguments [40]. This representation has the following issues for the software to circuit transformation: (a) The block arguments are *values* with no producer. (b) The branches collect the outgoing values, but these values are disconnected from the successor blocks. The former issue makes pattern rewriting—the standard method for converting between dialects—a poor fit for this conversion, as there is no operation to match. The latter issue makes locating the inputs to the multiplexers unnecessarily complex: unlike an LLVM ϕ node, where the input values from the other BBs are directly accessible, to obtain the same information, we need to first locate the parent BB, the branch, and then the values fed into the branch.

Dynamatic uses MLIR's pattern rewriting to carry out this conversion. We acknowledge that *this solution is not ideal*

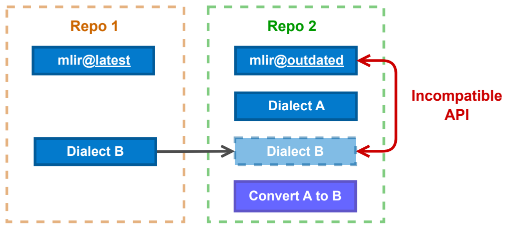


Figure 3. Integrating two MLIR projects is challenging.

since the rewrite must operate on the entire function (which defeats the purpose of the local rewrite rules).

4.3 Have We Solved the Software Segmentation Problem?

MLIR promises to resolve the software fragmentation problem by allowing the compiler projects to reuse each other's transformation passes. This might be true within the official MLIR repository, as the maintainers have a coherent view on how the segments would work together. However, many projects are not inside the upstream MLIR project, for example, CIRCT [41], Polygeist [42], Dynamatic [4], MLIR-AIE [43], etc. Can we smoothly integrate these projects?

Case study 4: implement a transformation pass for dialects maintained in two GitHub repositories. Figure 3 describes simplified architectures of two MLIR-based projects. These projects implement two different dialects, A and B. Suppose that we want to implement a new MLIR pass that converts dialect A to B. It is conceptually infeasible if the implementations of dialects A and B are dependent on two different LLVM projects with mismatching APIs (compilation error in either MLIR versions).

Dynamatic implements a custom translation pass that converts the handshake dialect into the XLS MLIR dialect [44]. Since it is impractical for us to constantly update and synchronize Dynamatic's LLVM version with XLS (which is updated daily), this feature has a risk of getting outdated very quickly.

4.4 Do We Have a Good HLS C Frontend that Encourages People to Build New HLS Tools?

An HLS *frontend* converts C code to IR, applies IR optimizations, and annotates important information for circuit generation. Is there a C frontend for MLIR that satisfies all these requirements?

Case study 5: The status of C-to-MLIR conversion. Some projects convert C to MLIR: *Polygeist* converts C AST to MLIR SCF dialect; the *CIR project* in Clang converts C AST to a CIR dialect (a dialect that models the C semantics). These solutions alone do not provide fine-grained IR optimizations, making them non-competitive with LLVM-based HLS compiler frontends.

Case study 6: The reliance on LLVM. Since LLVM provides a set of powerful and well-tested IR optimizations, most MLIR-based projects eventually lower their custom dialect to LLVM IR to benefit from LLVM IR optimizations. The ability to exploit the LLVM IR removes the incentive to implement the same optimizations in MLIR. Unfortunately, it also introduces all LLVM limitations into the compiler pipeline that MLIR aimed to avoid in the first place.

Since the existing MLIR C front-ends are not competitive with LLVM-based ones, Dynamatic is therefore still reliant on the LLVM IR for this process. We have to use custom workarounds for performing memory analysis and infer the array sizes from the original C code—these workarounds, difficult to implement in the rigid LLVM framework, would fit naturally in the MLIR compiler flow.

5 Beyond HLS

Although we have here focused on our particular use case and HLS experiences, we believe that the issues above are general and go beyond just HLS MLIR projects: (1) Both software and hardware compilers routinely exploit edge-specific information (e.g., memory dependencies, branch probabilities) and may suffer MLIR's limited expressiveness discussed in Section 4.1. (2) The SSA challenge of Section 4.2 is primarily hardware-oriented; yet, the lack of ϕs may complicate IR optimizations and analyses based on use-def chains and data dependencies in software compilers as well. (3) The software segmentation problem of Section 4.3 is perfectly general and applies to any MLIR project. (4) Any compiler flow that uses C as the input language suffers from the limitations discussed in Section 4.4.

We therefore believe that addressing these concerns would benefit the general MLIR community.

6 Conclusion

We discussed some major features of MLIR that lead to fragile workarounds in Dynamatic. We look forward to discussing the lessons learned and the future development of MLIR-based HLS projects with the LATTE community.

References

[1] *http://www.llvm.org*. The LLVM Compiler Infrastructure. 2018. URL: http://www.llvm.org.

[2] *Vivado High-Level Synthesis*. Xilinx Inc. 2018. URL: http://www.xilinx.com/products/%20design-tools/vivado/integration/esl-design.html.

[3] 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* 13.2 (Sept. 2013), 24:1–24:27.

[4] *Dynamatic*. EPFL-LAP. URL: https://github.com/EPFL-LAP/dynamatic/tree/999dc3ce2fb95eac1dd39cad441fbdf6b8389aee.

[5] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. "Dynamatic: From C/C++ to Dynamically Scheduled Circuits". In: *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, Feb. 2020, pp. 1–10. URL: https://doi.org/10.1145/3373087.3375391.

[6] *Catapult High-Level Synthesis and Verification*. Siemens EDA. 2026. URL: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.

[7] Fabrizio Ferrandi et al. "Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications". In: *Proceedings of the 58th ACM/IEEE Design Automation Conference*. San Francisco, CA, 2021, pp. 1327–1330. URL: https://doi.org/10.1109/DAC18074.2021.9586110.

[8] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *Proceedings of the 19th IEEE/ACM International Symposium on Code Generation and Optimization*. Seoul, Korea, 2021, pp. 2–14. URL: https://doi.org/10.1109/CGO51591.2021.9370308.

[9] *Multi-Level IR Compiler Framework (MLIR)*. LLVM Project. 2020. URL: https://mlir.llvm.org/.

[10] Jordi Cortadella, Marc Galceran-Oms, and Mike Kishinevsky. "Elastic systems". In: *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign*. July 2010, pp. 149–58.

[11] Lana Josipović, Radhika Ghosal, and Paolo Ienne. "Dynamically Scheduled High-level Synthesis". In: *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, Feb. 2018, pp. 127–36. URL: https://doi.org/10.1145/3174243.3174264.

[12] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. "Speculative Dataflow Circuits". In: *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, Feb. 2019, pp. 162–71.

[13] Lana Josipović, Philip Brisk, and Paolo Ienne. "An Out-of-Order Load-Store Queue for Spatial Computing". In: *ACM Transactions on Embedded Computing Systems* 16.5s (Sept. 2017), 125:1–125:19. URL: https://doi.org/10.1145/3126525.

[14] Ayatallah Elakhras et al. "Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits". In: *Proceedings of the 32nd International Symposium on Field-Programmable Gate Arrays*. Monterey, CA, Mar. 2024, pp. 44–54. URL: https://doi.org/10.1145/3626202.3637556.

[15] Lana Josipović et al. "Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs". In: *Proceedings of the IEEE International Conference on Field Programmable Technology*. Tianjin, China, Dec. 2019, pp. 197–205. URL: https://doi.org/10.1109/ICFPT47387.2019.00031.

[16] Lana Josipović et al. "Buffer Placement and Sizing for High-Performance Dataflow Circuits". In: *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, Feb. 2020, pp. 186–96. URL: https://doi.org/10.1145/3373087.3375314.

[17] Jianyi Cheng et al. "Combining Dynamic & Static Scheduling in High-Level Synthesis". In: *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, Feb. 2020, pp. 288–98.

[18] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. "Synthesizing General-Purpose Code into Dynamically Scheduled Circuits". In: *IEEE Circuits and Systems Magazine* 21.2 (Second quarter 2021), pp. 97–118.

[19] Lana Josipović et al. "Resource Sharing in Dataflow Circuits". In: *Proceedings of the 30th IEEE Symposium on Field-Programmable Custom Computing Machines*. New York, May 2022, pp. 1–9. URL: https://doi.org/10.1109/FCCM53951.2022.9786084.

[20] Jianyi Cheng et al. "Dynamic Inter-Block Scheduling for HLS". In: *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, Aug. 2022, pp. 243–52.

[21] Ayatallah Elakhras et al. "Unleashing Parallelism in Elastic Circuits with Faster Token Delivery". In: *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, Aug. 2022, pp. 253–61. URL: https://doi.org/10.1109/FPL57034.2022.00046.

[22] Carmine Rizzi et al. "A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits". In: *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, Aug. 2022, pp. 375–83.

[23] Jiantao Liu, Carmine Rizzi, and Lana Josipović. "Load-Store Queue Sizing for Efficient Dataflow Circuits". In: *Proceedings of the 21st International Conference on Field-Programmable Technology*. Hong Kong, Dec. 2022, pp. 1–9. URL: https://doi.org/10.1109/ICFPT56656.2022.9974425.

[24] Ayatallah Elakhras et al. "Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits". In: *Proceedings of the 31st International Symposium on Field-Programmable Gate Arrays*. Monterey, CA, Feb. 2023, pp. 39–45. URL: https://doi.org/10.1145/3543622.3573050.

[25] Jiahui Xu et al. "Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking". In: *Proceedings of the 31st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, Feb. 2023, pp. 27–37. URL: https://doi.org/10.1145/3543622.3573196.

[26] Carmine Rizzi, Andrea Guerrieri, and Lana Josipović. "An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits". In: *Proceedings of the 60th Design Automation Conference*. San Francisco, CA, July 2023, pp. 1–6. URL: https://doi.org/10.1109/DAC56929.2023.10247686.

[27] Jiahui Xu and Lana Josipović. "Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification". In: *Proceedings of the 42nd International Conference on Computer-Aided Design*. San Francisco, CA, Oct. 2023, pp. 1–9. URL: https://doi.org/10.1109/ICCAD57390.2023.10323796.

[28] Hanyu Wang, Carmine Rizzi, and Lana Josipović. "MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization". In: *Proceedings of the 42nd International Conference on Computer-Aided Design*. San Francisco, CA, Nov. 2023, pp. 1–9. URL: https://doi.org/10.1109/ICCAD57390.2023.10323639.

[29] Jiahui Xu and Lana Josipović. "Suppressing Spurious Dynamism of Dataflow Circuits Via Latency and Occupancy Balancing". In: *Proceedings of the 32nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, Mar. 2024, pp. 188–98. URL: https://doi.org/10.1145/3626202.3637570.

[30] Andrea Guerrieri et al. "DynaRapid: Fast-Tracking from C to Routed Circuits". In: *Proceedings of the 34th International Conference on Field-Programmable Logic and Applications*. Turin, Italy, Sept. 2024, pp. 24–32. URL: https://doi.org/10.1109/FPL64840.2024.00014.

[31] Jiantao Liu et al. "Fast Switching Activity Estimation for HLS-Produced Dataflow Circuits". In: *Proceedings of the 34th International Conference on Field-Programmable Logic and Applications*. Turin, Italy, Sept. 2024, pp. 118–125. URL: https://doi.org/10.1109/FPL64840.2024.00025.

[32] Jiahui Xu and Lana Josipović. "CRUSH: A Credit-Based Approach for Functional Unit Sharing in Dynamically Scheduled HLS". In: *Proceedings of the 30th International Conference on Architectural Support for Programming Languages and Operating Systems*. Rotterdam, The Netherlands, Apr. 2025, pp. 249–263. URL: https://doi.org/10.1145/3669940.3707273.

[33] Ayatallah Elakhras et al. "ElasticMiter: Formally Verified Dataflow Circuit Rewrites". In: *Proceedings of the 30th International Conference on Architectural Support for Programming Languages and Operating Systems*. Rotterdam, The Netherlands, Apr. 2025, pp. 293–308. URL: https://doi.org/10.1145/3676641.3715993.

[34] Mathias Bouilloud, Lana Josipović, and Wayne Luk. "Resource and Phase Awareness for Dynamically Scheduled High-Level Synthesis". In: *Proceedings of the 15th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. Kumamoto, Japan, May 2025, pp. 14–24. URL: https://doi.org/10.1145/3728179.3728194.

[35] Shun Katsumi, Emmet Murphy, and Lana Josipović. "EagerlyElastic: Correct-by-Construction Eager Execution in Dynamically-Scheduled HLS". In: *Proceedings of the 34th International Symposium on Field-Programmable Gate Arrays*. Seaside, CA, Feb. 2026, pp. 103–113. URL: https://doi.org/10.1145/3748173.3779196.

[36] Rouzbeh Pirayadi et al. "Out with LSQs: Custom Circuits for Memory Access Reordering in Dynamic HLS". In: *Proceedings of the 34th International Symposium on Field-Programmable Gate Arrays*. Seaside, CA, Feb. 2026, pp. 92–102. URL: https://doi.org/10.1145/3748173.3779204.

[37] Yann Herklotz et al. "Graphiti: Formally Verified Out-of-Order Execution in Dataflow Circuits". In: *Proceedings of the 31st International Conference on Architectural Support for Programming Languages and Operating Systems*. To appear. Pittsburgh, Mar. 2026.

[38] Lana Josipović et al. *Dynamatic Reloaded: An MLIR-Based Dynamically Scheduled HLS Compiler*. Tutorial at FPGA '24. Mar. 2024. URL: https://www.isfpga.org/past/fpga2024/workshops-tutorials/.

[39] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[40] *MLIR Rationale: Block Arguments vs PHI nodes*. LLVM Project. 2026. URL: https://mlir.llvm.org/docs/Rationale/Rationale/#block-arguments-vs-phi-nodes.

[41] *https://github.com/llvm/circt*. CIRCT IR Compiler and Tools. 2020. URL: https://github.com/llvm/circt.

[42] William S. Moses et al. "Polygeist: Raising C to Polyhedral MLIR". In: *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques*. Atlanta, GA, 2021, pp. 45–59.

[43] *IRON API and MLIR-based AI Engine Toolchain*. AMD. 2026. URL: https://github.com/Xilinx/mlir-aie.

[44] *XLS: Accelerated HW Synthesis*. Google, Inc. URL: https://github.com/google/xls.