MLIR: Scaling Compiler Infrastructure for Domain Specific Computation [1]

Federico Bruzzone,[1] PhD Candidate

Milan, Italy – 18 March 2026

[1]ADAPT Lab – University of Milan,
Website: federicobruzzone.github.io,
Github: github.com/FedericoBruzzone,
Email: federico.bruzzone@unimi.it
Slides: TODO

MLIR: Multi-Level Intermediate Representation

Part of the LLVM project, the MLIR is a novel approach to building **reusable**, **modular**, and **extensible** compiler infrastructure.

MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building **domain specific compilers**, and aid in connecting existing compilers together.

Why another compiler infrastructure?

Although the *one size fits all* approach of traditional compilers (e.g., LLVM [2] and JVM [3]) has been successful for general-purpose programming, it has shown limitations in the context of domain-specific applications.

Many problems are better modeled at a **higher-** or **lower-level abstraction** — e.g., source-level static analysis of C++/Rust is difficult on LLVM IR.

Hence, many languages and frameworks developed their own intermediate representations (IRs) to leverage the **semantic information** of their domain — including TensorFlow's XLA HLO, PyTorch's Glow, Rust's MIR, Swift's SIL, Clang's CIL, and so on.

While domain-specific IRs are well-understood, their *high engineering costs* often lead to compromised infrastructure quality. This results in *suboptimal compilers* plagued by bugs, latency, and a poor debugging experience [1].

MLIR to the rescue

MLIR directly addresses these issues by making it **cheap** to design and **introduce** new abstraction layers.

It achieves this by:

- standardizing the Static Single Assignment (SSA)-based IR data structures,
- providing a declarative system for defining IR *dialects*, and
- providing a wide range of common infrastructure including documentation, parsing and printing logic, location tracking, multithreaded compilation support, pass management.

Design Principles

- **Parsimony**: Apply *Occam's razor* to builtin semantics, concepts, and programming interface. Specify invariants once, but verify correctness throughout $\implies$ *extensibility*.
- **Traceability**: Retain rather than recover information. Declare rules and properties to enable transformation, rather than step wise imperative specification $\implies$ *composability*.
- **Progressivity**: Premature lowering is the root of all evil. Beyond representation layers, allow multiple transformation paths that lower individual regions on demand $\implies$ *reusability*.

Little Builtin, Everything Customizable *[Parsimony]*

- The system is based on a minimal number of fundamental concepts, leaving most of the intermediate representation fully **customizable**.
- A handful of abstractions—types, operations and attributes—should be used to express *everything else*, allowing fewer and more consistent abstractions that are easy to **comprehend**, **extend**, and **adopt**.
- A success criterion for customization is the possibility to express a diverse set of abstractions including **ML graphs**, ASTs, mathematical abstractions such as **polyhedral**, CFGs and instruction-level IRs such as **LLVM IR**, without hard-coding concepts.

SSA and Regions *[Parsimony]*

SSA [4] makes dataflow analysis *simple* and *sparse*. However, while many existing IRs use a flat, linearized CFG, representing higher level abstractions push introducing **nested regions**[2] as a first-class citizen.

[2] A region is a single-entry, multi-exit CFG that can be nested inside an operation. It is a generalization of the concept of basic blocks and allows for more flexible control flow representation.

Maintain Higher-Level Semantics *[Progressivity]*

Declaration and Validation *[Parsimony|Traceability]*

Source Location Tracking [*Traceability*]

Thank You!

Bibliography

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

[2] 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.*, 2004, pp. 75–86.

[3] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Addison-wesley, 2013.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991, doi: 10.1145/115372.115320.