xDSL: Sidekick Compilation for SSA-Based Compilers

Mathieu Fehr
University of Edinburgh
Edinburgh, United Kingdom
mathieu.fehr@ed.ac.uk

Michel Weber
ETH Zurich
Zurich, Switzerland
michel.weber@inf.ethz.ch

Christian Ulmann*
ETH Zurich
Zurich, Switzerland
christian.ulmann@inf.ethz.ch

Alexandre Lopoukhine
University of Cambridge
Cambridge, United Kingdom
sasha.lopoukhine@cl.cam.ac.uk

Martin Paul Lücke
University of Edinburgh
Edinburgh, United Kingdom
martin.luecke@ed.ac.uk

Théo Degioanni†
ENS Rennes
Rennes, France
theo.degioanni@ens-rennes.fr

Christos Vasiladiotis
University of Edinburgh
Edinburgh, United Kingdom
c.vasiladiotis@ed.ac.uk

Michel Steuwer
Technische Universität Berlin
Berlin, Germany
michel.steuwer@tu-berlin.de

Tobias Grosser
University of Cambridge
Cambridge, United Kingdom
tobias.grosser@cst.cam.ac.uk

Abstract

Traditionally, compiler researchers either conduct experiments within an existing production compiler or develop their own prototype compiler; both options come with trade-offs. On one hand, prototyping in a production compiler can be cumbersome, as they are often optimized for program compilation speed at the expense of software simplicity and development speed. On the other hand, the transition from a prototype compiler to production requires significant engineering work. To bridge this gap, we introduce the concept of **sidekick compiler frameworks**, an approach that uses multiple frameworks that interoperate with each other by leveraging textual interchange formats and declarative descriptions of abstractions. Each such compiler framework is specialized for specific use cases, such as performance or prototyping. Abstractions are by design shared across frameworks, simplifying the transition from prototyping to production. We demonstrate this idea with xDSL, a sidekick for MLIR focused on prototyping and teaching. xDSL interoperates with MLIR through a shared textual IR and the exchange of IRs through an IR Definition Language. The benefits of sidekick compiler frameworks are evaluated by showing on three use cases how xDSL impacts their development: teaching, DSL compilation, and rewrite system prototyping. We also investigate the trade-offs that xDSL offers, and demonstrate how we simplify the transition between frameworks using the IRDL dialect. With sidekick compilation, we envision a future in which engineers minimize the cost of development by choosing a framework built for their immediate needs, and later transitioning to production with minimal overhead.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: compilation frameworks, intermediate representations, interchange formats

ACM Reference Format:
Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Paul Lücke, Théo Degioanni, Christos Vasiladiotis, Michel Steuwer, and Tobias Grosser. 2025. xDSL: Sidekick Compilation for SSA-Based Compilers. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3696443.3708945

1 Introduction

When conducting compiler research, one can either embed a new idea in a preexisting compiler or develop an entirely new prototype compiler intended to show a speedup or other benefits. While writing a prototype may be more flexible in the short term, it comes with several drawbacks. First, a new prototype requires the reimplementation of many features, such as IR data structures, a parser and printer for the IR textual format, or generic passes such as dead code elimination. Later on, if the prototype is promising, porting it to a production compiler requires significant work, often a complete rewrite, and cannot always be done iteratively.

On the other hand, compiler frameworks reduce the cost of working in a production compiler by providing modular infrastructure that can easily be reused. For instance, MLIR [23] allows users to define their own abstractions, or

*Now at NextSilicon.
†Now at Nvidia.

CGO '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708945

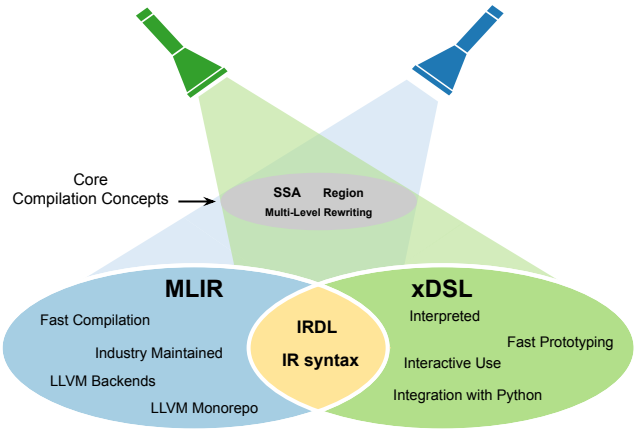

Figure 1. Compiler frameworks that share the same core compilation concepts can offer tailored projections of these concepts for specific use cases while being still compatible through IRs and declarative IR definitions.

to reuse abstractions from the ecosystem, which covers machine learning kernels [39], stencil computations [14], quantum computing [33], hardware design [5], and more. Due to the uniform representation that MLIR offers, abstractions can be freely combined, enabling compilation flows that combine domains, e.g., running ML inference in a database-style query [17]. Consequently, compiler frameworks facilitate compiler design and reduce prototyping costs.

However, while MLIR allows compiler experts to extend the compiler, it requires users to work in a setting that is designed for compile-time and runtime performance. As an industry-funded project that targets performance-focused production use cases, MLIR matches LLVM [22] in its choice of C++ as its implementation language, benefiting from a fast implementation, deployable across many target systems. Furthermore, MLIR's integration with LLVM goes beyond the implementation language since IRs are defined using LLVM's in-house TableGen language, LLVM's abstract data types are preferred over standard C++, and MLIR implements several low-level performance optimizations, e.g., to enable fast type equivalence checks via pointer comparison. While all these choices are justified, they mean that working with MLIR requires expertise in C++ and LLVM, costly development recompilation cycles, and complex build system maintenance – constraints that are hard to justify in some circumstances, in particular in teaching and research.

To facilitate other use cases such as prototyping, not only abstractions need to be connected in a modular way, but frameworks themselves. To that end, we propose the idea of ***sidekick compiler frameworks***, which are frameworks loosely coupled through the use of shared core compilation concepts (Figure 1) and aligned exchange formats for IRs and IR definitions. In particular, a sidekick framework can be interleaved at any point in the compilation pipeline with its base framework. By deliberately reusing the same core compilation concepts, like SSA and multi-level rewriting, and exchanging IR definitions between frameworks through a common format, a sidekick framework eases the transition between frameworks.

In this paper, we present xDSL, a sidekick compiler framework for MLIR written in Python. xDSL is standalone, but interacts with MLIR through a shared textual IR format, and the IRDL dialect, an MLIR-based meta-IR that expresses IR definitions as programs. To evaluate the benefits of xDSL as a sidekick compiler framework, we explore three compiler use cases that benefit from xDSL's Python-native implementation and show several statistics to compare xDSL and MLIR. Our analysis shows that bringing state-of-the-art MLIR concepts to new use cases results in a broader and better-connected compiler ecosystem that can cater to various novel workflows.

Our contributions are:

- The concept of a *sidekick compiler framework* coupled to a base framework via deliberately sharing compilation concepts (e.g., SSA-based IRs, nested regions, attributes) and compatible textual IRs (Sections 2 and 4).
- Three case studies that characterize workflows that benefit from sidekick compilation (Section 3).
- An encoding of IR definitions as SSA-based compiler IR (the IRDL dialect) for the exchange of IR definitions between our sidekick and base framework (Section 5).
- A comparison of several user-relevant metrics between xDSL and MLIR (Section 6).

2 Sidekick Compilation

A sidekick compiler framework can be used in a completely standalone way yet can be connected to a base compiler framework through a common IR exchange format. To achieve this, sidekick frameworks use the same core compilation concepts as the base framework, such as SSA-based IRs, or nested regions. A fundamental advantage of sidekick frameworks is that their implementation can be tailored to the needs of their target audience, e.g., by providing a simple implementation of the core IR to ease prototyping, or by providing a verified implementation to enable formal verification of the compiler. Finally, sidekick frameworks can share IR definitions through an additional exchange format, simplifying the port of code between the two frameworks.

We demonstrate the concept of sidekick compilation with xDSL, a Python-native compiler framework that interoperates with MLIR. xDSL offers a standalone compiler framework that targets developers who use Python for their main workflows, or want to quickly prototype new compiler ideas or abstractions. The dynamic nature of Python also allows the use of xDSL in new environments such as Jupyter Notebooks [19]. The core novelty of xDSL is its coupling with the base compiler MLIR (Figure 2). xDSL couples to MLIR by mirroring its core IR structure and its textual IR.

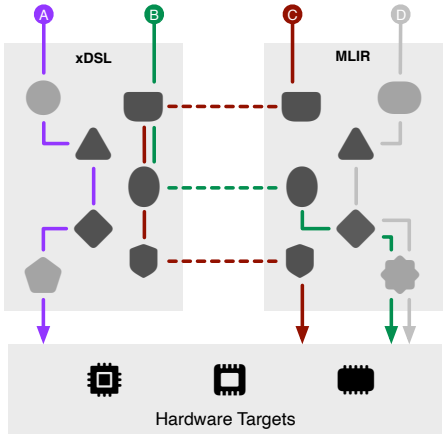

Figure 2. Dialect definitions (gray shapes in xDSL and MLIR) can be shared between xDSL and MLIR (dark gray shapes exist in both). While the manipulation of IRs is done with internal data structures inside both projects (solid lines), the sharing of IRs and IR definitions across frameworks is done through a shared textual representation (dashed lines). This enables the exploration of new workflows in the compiler space: A python-native development of end-to-end compilers, B pairing high-level DSLs with low-level compilers, and C extending low-level compilers to explore new compiler design ideas.

A sidekick compiler's ability to exchange IR definitions and programs with its base compiler enables several new workflows, some of which we highlight briefly. The most straightforward workflow A uses xDSL as a standalone compiler framework to implement a self-contained compiler. The core property of such a workflow is that it stays entirely within Python, enabling developers to run the compiler on any Python-supported platform, iterate quickly by extending the compiler at runtime, or integrate Python libraries with ease into the compilation flow. Interestingly, even an independently used sidekick still can leverage IR definitions that were initially developed in the base compiler. xDSL also enables workflows B that yield a full DSL compiler by combining a domain-specific front-end implemented in Python with the hardware targets available in MLIR. The use of MLIR and LLVM offers additional low-level optimizations, powerful register allocation and instruction selection, as well as infrastructure for targeting the latest hardware accelerators. Furthermore, xDSL allows workflows C where xDSL is placed into a pre-existing compilation flow of the baseline compiler. Such workflows make it possible to prototype new compilation approaches and ideas from Python, for example, new rewriting systems. By porting IR from MLIR to xDSL, using the xDSL-based prototype, and then going back to MLIR, one can show the potential of new approaches using an early Python-based prototype. While these examples correspond to the use cases shown later (Section 3), they are not exhaustive, and we expect other future uses.

One of the core novelties that enables sidekick compilation in xDSL is its ability to share IR definitions with MLIR. Expanding on MLIR's declarative IR definition language IRDL [12], we made it possible to share IR definitions between xDSL and MLIR by encoding IR definitions with a novel SSA-based meta-IR, the IRDL dialect, that can be exchanged between compilers like any other program. xDSL and MLIR can both translate their IR definitions into the IRDL dialect and import IRDL dialect definitions to instantiate externally provided IRs. Implementing this is relatively easy. Both xDSL and MLIR share the same definition of the IRDL dialect meta IR. As we can translate programs between the two frameworks, we can also translate IR definitions and, therefore, use IRs defined in MLIR from xDSL and vice versa. This gives access to the existing ecosystem of MLIR dialects and also makes importing IRs from xDSL into MLIR possible.

While a sidekick compiler such as xDSL does not share transformations with MLIR, there is ongoing work to connect xDSL with MLIR's PDL dialect, a dialect to define and reason about IR rewrites. Using PDL, we expect to eventually be able to port rewrites from one compiler to the other, making the rewrites themselves less implementation-dependent and potentially allowing the transfer of increasingly complete compilation flows across compiler boundaries.

3 Use Cases for SideKick Compilation

We demonstrate three use cases by discussing the users and their respective needs, how existing workflows address, or fail to address, those needs, and how xDSL facilitates the uses case or even enables them in the first place. We used xDSL to teach compilation at DoubleBlind University, and prototyped a new rewriting engine for the multi-level rewriting approach. We also present its usage in an HPC DSL compiler that leverages MLIR's low-level optimizations to reach state-of-the-art performance.

3.1 Use Case 1: Teaching Compilation with ChocoPy

While most compilers are written by professionals and for production use, implementing a compiler is also a great way to teach compilation concepts. However, writing an SSA-based compiler from scratch is a complex task, and using frameworks helps focus on the compilation concepts rather than the implementation details of the data structures in typical SSA compilers. We used xDSL for two years in the compilation class of DoubleBlind University to teach around 200 students. We tasked students with implementing a compiler for ChocoPy [31], a subset of Python designed to teach compilation to students. Students must implement a parser, a type checker, optimizations at multiple abstraction levels, and a lowering to RISC-V.

User. While typical compiler engineers often have high-end computers capable of quickly building and running production-quality compilers, students may have slower computers with diverse architectures and operating systems. Students often have less programming experience and less experience with package managers and build systems. Finally, most students, especially in introductory classes, may not pursue compiler work in the future, and thus investing much time in a framework may not be valuable for them.

Needs. Students have different needs than typical compiler engineers. Because of consumer hardware, compiling large frameworks is often impractical, and long incremental building times are a big source of frustration. The installation needs to be simple and quick since many students are unfamiliar with build systems. Optimally, students should be able to install and start trying out the framework during a single lab session (2 hours). Finally, frameworks that are not portable across architectures and operating systems are known sources of problems. A compiler implemented as a lecture project is not required to be as fast as industry compilers. Thus, frameworks with longer compilation times than state-of-the-art compilers are not an issue.

Existing Workflows. One existing workflow would be to not provide any framework to the compiler class and, instead, ask students to implement their own data structures. This is, for instance, the approach taken by the original ChocoPy compiler class. We argue that this solution is not optimal for the students, since they have to implement a lot of boilerplate code that is unrelated to the compilation techniques they are learning. Thus, the time required to set up an end-to-end compilation flow already consumes most of the available time in a lecture project setting. Also, most of these compiler classes usually do not teach the important SSA representation because it is hard to write SSA compiler infrastructure.

Another possibility is using an existing compiler infrastructure. Some compiler classes use LLVM, but LLVM can only be used as a mid-level IR and cannot be used for high-level or low-level IRs. While MLIR has the core compilation concepts that interest us, it is both complex and slow to install. Also, iterating on a compiler is expensive with MLIR, especially with low-end machines. Finally, MLIR has a steep learning curve due to its low-level nature, and thus students may lose significant time understanding the framework.

The xDSL Approach. We argue that using xDSL is the best solution for the students. Students are able to install the framework and start using it in a single lab, which means that we can directly help them with the core part of the coursework. Also, since the framework is not optimized for compile-time performance, it is significantly easier for students to express what they want, especially for students with less programming experience. Finally, since xDSL is written in Python, which does not need to be compiled, there is no time spent waiting for recompilation after a single change, and thus students can iterate on their compiler much faster.

3.2 Use Case 2: Designing a DSL Compiler

Many domain-specific compilers reimplement a lot of common infrastructure for parsing/printing, optimizations and, more importantly, domain abstractions. This redundant development leads to isolation between different projects, which negates many of the benefits gained in usability and productivity. To address this issue, projects such as CVM [29] for databases, CIRCT for hardware and MLIR [23] for deep learning libraries and general compilation have emerged, providing extensible and composable IRs. These approaches aim to reduce reimplementation and promote shared infrastructure. However, optimizing the architecture for such a unified compilation stack poses significant challenges, often requiring multiple iterative design cycles to achieve optimal performance and functionality across diverse domains.

xDSL was used by another research group to develop a compilation stack for the HPC domain, specifically focusing on providing stencil abstractions. This served as a substantial shared foundation for two HPC stencil-DSL compilers and a state-of-the-art research DSL compiler for climate modelling [1, 14, 28]. The project implemented a stencil dialect, extending concepts found in the Open Earth Compiler [14], and complementary dialects that express the parallel and distributed computation and memory concepts of typical HPC compute environments. During machine code generation, xDSL leverages MLIR to lower and optimize for the intended hardware architecture, generating high-performance executables. Our tool allowed fast prototyping over the design of these concepts, which, apart from enabling infrastructure sharing and competitive performance, also culminated in contributing a mature dialect back to the MLIR ecosystem.[1]

User. The user is an HPC scientist, though the concepts apply to many similar domains. Such users can be slowed down by complex build systems and frameworks that require significant time to get started. Thus, they gravitate towards frameworks that minimize cognitive resources expended on low-level implementation intricacies, thereby maximizing focus on domain-specific problem-solving and innovation.

For this use case, the most crucial need is a low engineering effort for defining and modifying dialects, as it allows efficient understanding and exploration of the design space for domain-specific abstractions. When working within their domain, experts prefer to avoid concerns with low-level details like memory allocation, as operating at a higher level of abstraction facilitates the translation of thoughts into code.

This preference is reinforced by the ability to rapidly prototype, which is enabled by the absence of complicated compilation systems and the use of interpreted languages. Additionally, demonstrating the performance potential of generated machine code plays a crucial role in the scientific validation and assessment of these domain frameworks. In this context, Python has already emerged as a particularly suitable tool,

[1] commit: b334664f9f3a098b6f3fd9cfd17b856a9edfe446

offering a combination of simple, high-level code and freedom from complex build systems. This makes xDSL more appropriate for domain experts than low-level languages like C++, as it aligns with their need for rapid iteration, high-level abstraction, and focus on domain-specific logic while still providing a path to high-performance implementations.

Existing Workflows. While there is a need to unify domain-specific compilers, they usually remain secluded in their own domain and generate source code leveraging general-purpose compilers to produce binaries. Separate infrastructures make cross-domain compilation hard, leading to replicated optimizations and abstractions (e.g., stencil IRs).

MLIR enables the fusion of domain-specific with general-purpose compilers, removing the aforementioned code duplication. However, while working with MLIR, a developer needs to be mindful of low-level details. This complexity incurs a cost when refactoring code with new design decisions, rectifying errors, and even during initial setup. The overhead appears both at the implementation level (e.g., creating a new dialect requires non-trivial changes in at least two or three files) and conceptually, presenting a steep learning curve to understand the ecosystem's architecture.

The xDSL Approach. Although xDSL does not eliminate the need to understand MLIR concepts, its distribution via PyPi simplifies the process of getting started and facilitates experimentation. Moreover, being implemented in native Python makes it more easily usable by other Python DSL-based projects, which have established a strong presence in the HPC ecosystem [4, 28]. Therefore, users can write in a high-level language without having to think about low-level implementation details when implementing domain-specific optimizations. The choice of Python also makes iterating on design decisions much faster than a C++-based flow.

xDSL has leveraged MLIR's, and hence LLVM's, low-level optimizations and code generation to produce optimized binaries. It has been used to merge the backends of three HPC DSLs – Devito [28], the Open-Earth Compiler [14], and Psyclone [2] – by providing a set of common abstractions for them to target within xDSL [4]. The evaluation of this prototype over representative stencil workloads, exhibited comparable and even improved performance over the original individual DSLs, while simultaneously broadening the hardware targets. This work has contributed back to MLIR, a mature and evolved dialect design of a standardized message-passing communication protocol used widely in the HPC ecosystem. The MPI use case validates our sidekick approach, demonstrating ecosystem consolidation rather than fragmentation. By prototyping in xDSL and porting to MLIR, compiler writers can seamlessly integrate both ecosystems, promoting interoperability in the compiler landscape.

3.3 Use Case 3: Prototyping New MLIR Features

While most changes to compilers are incremental, such as adding new transformations, compiler researchers are also concerned with re-imagining the core design of compilers. We used xDSL to prototype a new approach to a rewriting system for MLIR, enabling declarative, composable and controllable rewrites inspired by Elevate [15]. This approach allows for expressing optimizations using simple abstractions that are composed to form complex rewrites. The rewriting system leverages an immutable IR, which forbids arbitrary mutations and enables backtracking with low memory cost. Making a mutable IR, the de facto standard, into immutable requires invasive changes in modern compilers.

User. The users are compiler researchers who aim to design the next-generation compilers by extending or modifying core design aspects. They are familiar with LLVM and MLIR and understand the codebases well and work individually or in a small research group using a wide range of hardware, from company-provided laptops and high-end desktops up to super-computing hardware. They want to prototype and evaluate new ideas quickly and, if beneficial, publish and contribute them back to industry compilers (e.g., via LLVM and MLIR).

Needs. The number one priority of the users is to prototype ideas quickly and evaluate their feasibility. Researchers do not want to waste time engineering low-level details for an idea that is not beneficial. Furthermore, it is essential for them to iterate quickly and benchmark multiple designs against each other. Thus, a framework with quick build times is preferred. While observing the performance and memory trade-offs of different prototypes is important, it is rarely the aim to achieve production performance with the prototype. Finally, it is crucial to test a prototype with real-world programs and observe its results in an end-to-end compilation pipeline. Accordingly, it is imperative that a tight integration with MLIR is possible, such as integrating a prototype into an MLIR pass pipeline.

Existing Workflows. The existing approach is to experiment with the prototype design directly in MLIR using C++. This forces the user to split their effort between making design decisions for a prototype and managing low-level implementation details, such as maintaining existing storage layouts. These details often have to be revisited later while tweaking the prototype design and arguably put a strain on productivity, making it unsuitable for fast iteration. The burden of long build times on local consumer hardware is amplified when multiple prototypes have to be designed and compared. The focus of the MLIR framework leans heavily towards production performance rather than a clear structure for easy understanding and extensibility of its core concepts. This aggressive optimization for performance has to be considered constantly and leads to a rigid system with hard-to-understand design decisions and poor modifiability.

We implemented the envisioned rewriting system by investing four months and managed to design a working prototype with limited support for composing rewrites. However, it lacked the foundation of an immutable IR and thus

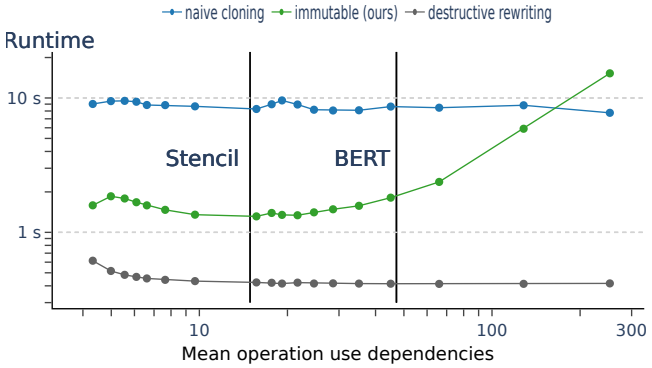

Figure 3. Our immutable approach has a runtime between naive cloning and destructive rewrites. For code such as climate stencils and BERT, we predict only a 4x cost for the benefit of immutability.

could not prevent arbitrary IR modifications or support backtracking. For this reason, it also exhibited several unsolved problems when rewriting complex programs. While the researcher is very familiar with the C++ programming language and the details of the MLIR implementation, a large portion of the time was spent on getting the template-based C++ typing correct with all required features to achieve a composable API interface. Furthermore, a lot of time was spent implementing features and tweaking the MLIR framework and making it usable for this use case, instead of the prototype. This includes multiple DSLs to ease access to complex MLIR APIs and combat the verbose nature of C++.

The xDSL Approach. xDSL exposes the main MLIR concepts and high-level design decisions in a Python interface. Thus, it enables the implementation and modification of core MLIR constructs while leveraging the productivity of Python. This makes experimentation with drastically different design approaches in the context of MLIR practical in the first place. For instance, modifying the core IR infrastructure to support dependent typing can be implemented following a number of approaches. Instead of modifying numerous C++ files per approach, with little opportunity for reuse between completely different approaches and handling a complicated build system, in xDSL, the core IR can be flexibly switched by extending or replacing one Python file. xDSL empowers the researcher to switch between different core IR designs using a flag to flexibly benchmark them.

In about eight weeks, we designed a first working prototype with xDSL offering equal capabilities to the C++ prototype. After four months, the time required with C++ for a basic prototype, the new prototype had significantly advanced, now backed by an immutable IR infrastructure with full backtracking support. The similarity in structure to the MLIR framework made the adoption of xDSL straightforward without prior experience. The API interface was easily adjustable to the current needs. With fewer requirements for managing low-level implementation details, it became possible to focus solely on the design of the prototype and iterate quickly. The close interaction of xDSL and MLIR made benchmarking the rewriting of real-world machine learning models practical. Our prototype shows the advantages of leveraging an immutable IR for a backtracking rewriting system and allows us to evaluate the trade-off in memory and processing speed when using a representation of IR that can efficiently recover a previous state in the compilation process. Figure 3 evaluates the approaches' rewriting time and memory consumption by varying the IR structure to be optimized, i.e., the number of uses of the mean operation. A mutable rewrite system that performs a naive cloning of the IR to support the backtracking pays a high overhead independent of the IR structure. In contrast, the performance of our immutable rewriting system heavily depends on the structure of the IR. The more uses the mean operation has, the less reused; hence, rewriting time and memory consumption increase. However, for the structure of real-world use cases, such as weather modelling stencil computations from the Open Earth Compiler [14] (left vertical line) or the BERT-small [9] (right vertical line) transformer model, our approach requires less rewriting time and consumes much less memory.

4 Sharing Core Compilation Concepts

One major property of a sidekick compiler framework is its ability to share core compilation concepts, and IR representation, with another framework. xDSL shares these concepts with MLIR, pairing SSA with block arguments and nested regions, which MLIR demonstrated to be an effective set of core abstractions. While the implementation and in-memory representation of these concepts differ, as we tailor our implementation to be simple and Python-native, differing from MLIR's performance-focused C++ implementation, our IR textual representation is compatible with MLIR's. This compatibility purely comes from sharing the same textual representation, and allows us to exchange IRs during the compilation of a program, which is essential to enable the workflows we discussed in Section 3. We now describe the core compilation concepts we share with MLIR.

4.1 Operations and Values

Operations are the core structure of the IR and represent both computations and control flow structures (e.g., loops). An operation consists of a name, a list of operands, a list of results, a dictionary of attributes (Section 4.2), and lists of regions and successors (Section 4.3). Each operation name defines a set of invariants over these structures, called the verifier, to ensure that the operation is well-formed. For instance, the `arith.addi` operation expects two operands and one result, all with matching integer types. Operations have a generic textual format (upper line in the following example) but can also be extended with a custom format (lower line) for conciseness and readability:

```
%res1 = "arith.addi"(%v1, %v2)
          : (i32, i32) -> i32
%res2 = arith.muli %v1, %v2 : i32
```

Operations are connected through SSA values, which represent runtime values, prefixed by % in the textual IR. Each SSA value has a single statically known definition, either as a result of an operation or as a block argument. Values are only used as operands to operations.

4.2 Attributes and Types

Attributes encode compile-time information (e.g., constants or types) and are used to parametrize both SSA values and operations. Types are special attributes that provide static constraints over SSA values. For instance, the `i32` type on a value encodes a signless 32-bit integer. Attributes attached to operations through their attribute dictionary encode operation parameters. For instance, the operation `arith.constant` with the named attribute `value = 42 : i32` encodes that the resulting SSA value is the integer 42 encoded in 32 bits:

```
%cst = "arith.constant"() {value = 42 : i32}
         : () -> i32
```

As with operations, attributes and types are not a fixed set but can be extended with user-defined attributes and types. However, attributes may be parametrized by arbitrary data (for example `42 : i32` being parametrized by an integer value and an integer type), and thus their textual representation is not generic.

This definition of attributes and types differs slightly from their original MLIR meaning. In MLIR, attributes and types are entirely disjoint, though there exists a `TypeAttr` attribute that encodes a type as an attribute. We made types a special case of attributes to simplify the design of the core language. Without the constraint of wanting to support MLIR syntax, we would not have defined types at all and allowed the use of any attribute as a type annotation.

4.3 Regions and Blocks

While operations represent computation and structures, they are insufficient to represent control flow graphs. Instead, structures that reason about control flow in a larger sense are needed. In xDSL, these correspond to blocks and regions.

A block is a sequence of operations that execute in order. Blocks are connected through their last operation (called the terminator), which specifies which block to jump to next. This forms a directed graph of blocks, called a Control Flow Graph (CFG). Blocks may have block arguments, which introduce SSA values, giving the IR a functional structure that has been shown to be equivalent to phi-nodes [36].

```
^b0(%c: i1):
  scf.cond_br %c, ^b1, ^b2
^b1: ...
^b2: ...
```

A region is wrapping a CFG, and is nested in an operation. In contrast to IRs like LLVM's, which require analysis passes [6], regions model nested control flow as first-class constructs to represent structures such as loops or conditionals. For instance, a `scf.if` operation, representing a conditional, contains two regions, and execute the region depending on the runtime value of its only operand. Once a region finish executing, control flow is returned to the operation, which may give control to another region it contains, or terminate. For instance, an `scf.for` (a for loop) may execute its region multiple times.

```
scf.if %cond {
  // True region
} else {
  // False region
}
```

4.4 Dialects

Operations and attributes that represent similar concepts are grouped in dialects, allowing separation of concerns. For instance, the `arith` dialect contains operations for simple arithmetic, and the `scf` dialect for operations with structured control flow, such as loops and conditionals. This separation of concerns allows multi-level compilation pipelines, which interleave domain-specific optimizations of dialects with progressive lowerings to lower-level dialects. Hence, multiple dialects can be used in the same program, allowing lowerings to only target parts of a program.

4.5 Different Implementations

Despite having compatible textual representations, the in-memory representation and API that both xDSL and MLIR provide differ. This choice is deliberate, as we aim to provide a simple Python implementation that makes development and experimentation easy. In contrast, MLIR aims to provide a high-performance C++ implementation that is optimized for production use. For example, MLIR data structures are heavily optimized so that a single operation fits in a cache line, and attributes are instantiated uniquely in memory to reduce their footprint.

MLIR's optimizations result in having multiple C++ classes to represent operations and attributes, spanning multiple large C++ files. While these optimizations are key to MLIR performance, they heavily complicate its core implementation, hindering experimentation and making it hard to understand the core concepts of its IR. In contrast, xDSL's implementation of operations and attributes is a single Python class each, making it easy to understand and experiment with the core concepts of the IR, enabling most of the use cases we discussed in Section 3.

```
Dialect cmath {
  Type complex {
    Parameters (elem: !AnyOf<f32, f64>) } }
                        ↓
irdl.dialect @cmath {
  irdl.type @complex {
    %is_f32 = irdl.is f32
    %is_f64 = irdl.is f64
    %is_float = irdl.any_of(%is_f32, %is_f64)
    irdl.parameters(%is_float)
} }
```

Figure 4. Our new IRDL dialect exposes the IRDL language definitions (top) as an IR program (bottom), which can be easily shared across compilers.

5 Defining and Sharing Dialect Definitions

While both xDSL and MLIR define dialects using their respective programming languages, efforts have been made in the MLIR project to make these abstractions more declarative through the IRDL language [12]. We leverage this work with the IRDL dialect, which uses an SSA representation to define MLIR abstractions as programs, allowing us to share one abstraction definition between xDSL and MLIR. We also provide a frontend for IRDL in Python to provide a better interface for defining abstractions in xDSL.

5.1 The IRDL Dialect: An IR for IR Definitions

The IRDL dialect is defined in MLIR and xDSL, allowing both frameworks to share dialect definitions the same way they share programs. IRDL programs define dialects, types, attributes, and operations using a small but expressive constraint engine derived from IRDL. Making IRDL a dialect allows one to easily embed it in any compiler framework offering SSA dialect infrastructure, such as MLIR or xDSL. As dialects are now input data for compiler infrastructures, IRDL-defined dialects inherit the introspectable, portable and transformable nature of any other IR program.

Dialects, operations, types, and attributes are defined using the `irdl.dialect`, `irdl.operation`, `irdl.type`, and `irdl.attribute` operations (Figure 4). Attributes are defined by constraining their parameters, and operations are defined with constraints on their operands, results, attributes, successors, and regions. In the IRDL dialect, each SSA value represents an attribute (or a type), and constraints over these attributes are expressed through IRDL dialect operations. For instance, `irdl.is`, constrains an attribute to be equal to the given attribute, and `irdl.any_of`, constrains an attribute to be one of the given attributes.

We worked with the MLIR maintainers to bring the IRDL dialect to the main MLIR repository. The current implementation can register new IRDL dialects at MLIR runtime, but does not yet support generating C++ definitions. Also, while the IRDL dialect already provides a way to register operations and attributes that can be defined purely declaratively, it does not yet provide escape hatches to the framework language for more complex constraints.

An implementation-agnostic concept. The sidekick compiler approach is challenging to put into practice when dialect definitions are deeply embedded within one compiler. Instead, the IRDL dialect represents dialect definitions in a compiler-agnostic manner. All sidekick compilers implementing an IRDL dialect-like registration endpoint can easily share dialect definitions, as long as they can translate their dialect definitions to the IRDL dialect. This approach allows sharing of the core concepts of the modeled dialects without having to reimplement them in each compiler.

5.2 PyRDL: Connecting IRDL to xDSL

xDSL dialects are registered using PyRDL, a Python-embedded DSL implementation of IRDL (Figure 6). PyRDL defines accessors, verifiers, and parser/printer functions for types, attributes, and operations. The EDSL is also type-safe, so Python type-checking tools will correctly understand the types of operand or attribute definitions. For instance, the Python type `ComplexType[f32]` is the attribute instantiated by `!cmath.complex<f32>`.

Operation and parametrized attribute definitions can be automatically translated back and forth to the IRDL dialect. The translation to the IRDL dialect is done using Python introspection, while the translation from the IRDL dialect to PyRDL is implemented as a Python script. In particular, Data attributes cannot be translated to the IRDL dialect, as they rely on user-defined Python data structures that cannot be understood by the declarative nature of the IRDL dialect. Similarly, C++ constraints and attributes with C++ parameters can be translated to PyRDL, but with a generic of any value.

5.3 A Shared Dialect Ecosystem

Using the translation from the IRDL dialect to PyRDL, we can use the entire MLIR ecosystem in xDSL. Combining the projects using MLIR, such as MLIR, Flang, or CIRCT, with several xDSL-based projects, leads to an ecosystem with a plethora of dialects and operations (Figure 5). While the number and name of operands, results, attributes, successors, and regions are always translated, the constraints defined in C++ rather than in IRDL are not. Instead, they are replaced by a generic constraint that accepts any value.

While operations can be translated from one framework to the other using the operation generic format, attributes (and thus also types) do not have generic format. MLIR does provide a declarative specification for attribute custom format, but it is not used by every type and attribute, and may embed arbitrary C++. Thus, to convert attributes used in programs, we need to manually implement a printer and a parser for each attribute definition. To quantify the number

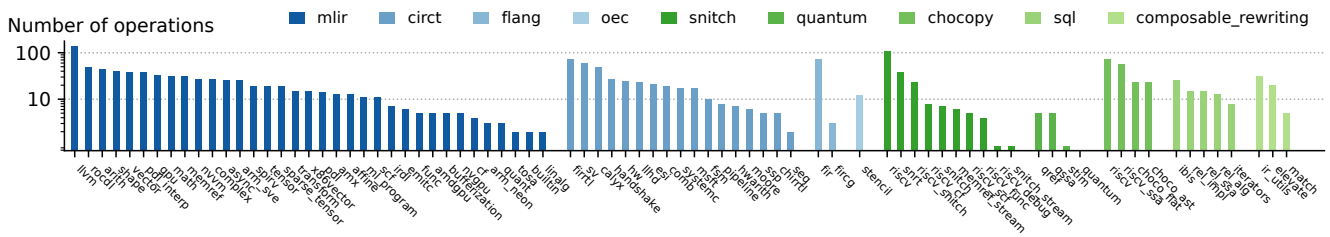

Figure 5. xDSL combines its 25 SSA dialects (521 ops) (green) with the 54 SSA dialects (1198 ops) defined in the Flang, CIRCT, and MLIR Core community repositories (blue).

```
T = TypeVar("T", bound=Union[f32, f64])
@irdl_attr_definition
class ComplexType(Generic[T], ParametrizedAttribute):
  name = "cmath.complex"
  elem: ParameterDef[T]
```

Figure 6. Defining dialects using a Python EDSL allows us to translate them back and forth to IRDL.

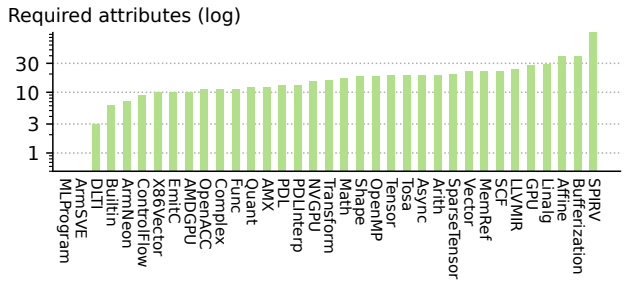

Figure 7. The full use of a dialect in xDSL requires few attributes.

of attributes that need to be ported to use a dialect fully, we count the number of distinct attributes used in the dialect test folder in the MLIR test suite (Figure 7). We find that most dialects only use a few distinct attributes and most dialects require less than 10 attributes to be fully usable in xDSL. Additionally, all dialects, except SPIRV, require less than 30 attribute definitions.

6 Compiler Design Space Characterization

To better understand the design-space trade-offs between xDSL and more traditional production frameworks, we compare against MLIR on multiple metrics relevant during compiler development. To that end, we not only compare both compilers based on their runtime, but also on the time taken to compile and install the compilers themselves. While xDSL is slower than MLIR for larger files, it performs similarly for smaller ones, like those used in testing. Furthermore, we show that xDSL uses significantly fewer resources to install and run after a change in the compiler. This comparison demonstrates that xDSL offers users a point in the compiler design space that prioritizes developer productivity over the performance of the compiler executable.

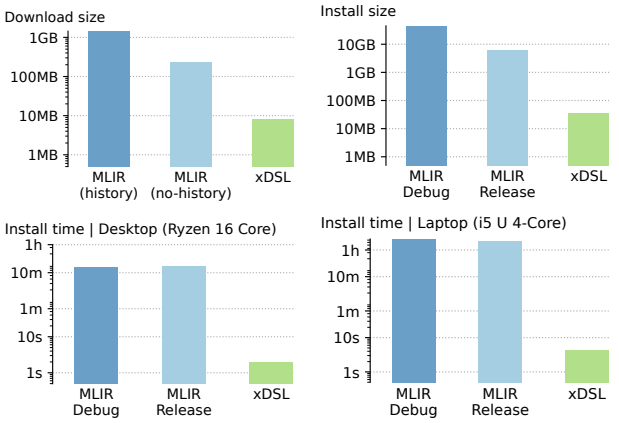

Figure 8. xDSL installation is lightweight, making it usable on low-end machines.

6.1 Startup and Build Times

Two important metrics that are often overlooked in a compiler framework are the costs of installing and running it for the first time, and running it after an incremental change. While these features rarely matter for end users, they do for compiler developers. We compare these metrics between xDSL and MLIR, both compiled in release mode, which enables compiler optimizations, and in debug mode, which does not enable optimizations and allows the use of debuggers. Both versions are compiled with Clang version 14.0. We compare these metrics on two devices that correspond to two potential users: A desktop using an AMD Ryzen 9 5950X 16-Core CPU, which is a relatively high-end CPU that could typically be used by a compiler engineer and a laptop with an Intel i5 10210U 4-Core CPU, which was given two years ago to PhD students at DoubleBlind University.

Startup time. First, we compare the startup time, i.e. the time necessary to run the test suites for the first time after downloading the repository. This is representative of the use of MLIR and xDSL since it is the only way to modify the abstractions and passes included in the projects. While installing xDSL is done by a local install with `pip`, installing MLIR is done by compiling it with `CMake`. On both machines, compiling MLIR requires two orders of magnitude more time than xDSL (Figure 8), taking almost 1 hour on the laptop

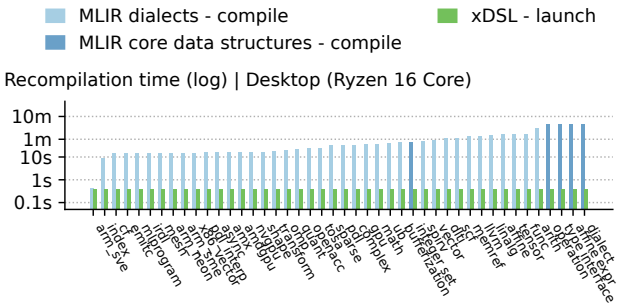

Figure 9. Recompiling MLIR after a single change is up to two orders of magnitude slower than launching xDSL, impacting prototyping and development efficiency.

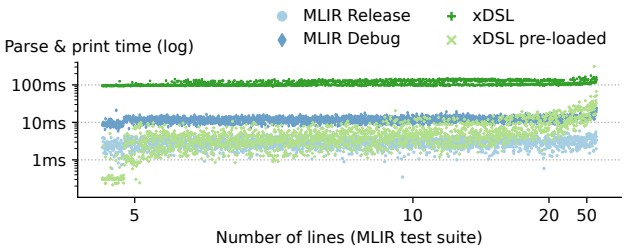

Figure 10. A pre-loaded xDSL parses and prints production test cases at a similar speed as MLIR compiled in Debug mode, making it a suitable alternative for testing a prototype.

and 10 minutes on the desktop, compared to the few seconds the xDSL setup needs on both machines. We observe that the compilation on the laptop is significantly slower than on the desktop as the laptop only has a few cores, and the compilation of MLIR can be heavily parallelized.

Download and installation size. Most users will install the projects manually on their machines, requiring to install around 2 MB for xDSL, and 1 GB for MLIR (which can be lowered to 100 MB by not downloading the entire git history). However, some users will instead opt for a pre-compiled version of MLIR, for instance, compiled in a docker container. While this reduces the cost of compiling MLIR, it still requires significant disk space. While the entire folder of xDSL, including a virtual environment for Python, requires less than 50 MB, the installation folder of MLIR is more than 4 GB in release mode and 26 GB in debug mode (Figure 8), resulting in long download times for MLIR binaries. Note that the installation size of MLIR is lower for users that only need a subset of the dialects or executables. xDSL' smaller installation size simplifies its distribution for many users and enables its use in cloud or web environments.

Incremental builds. Another interesting metric is the time taken to launch the compiler after a single change in its definition. This metric represents the daily use of the frameworks, as having a lower recompilation time is essential for rapid iteration of changes in the compiler. To measure this, we add a single space character at the end of a file containing a dialect or core data structure implementation and measure the time to launch. While MLIR requires a costly partial recompilation after a change, xDSL startup does not change, being the cost of loading the Python source files. On the desktop, the majority of dialects require over than 14 seconds of recompilation, up to 2 minutes, and some core data structures require more than 5 minutes of recompilation after a change (Figure 9). On the laptop, these can take more than 10 minutes. When iterating on dialects on low-to-middle-end hardware, MLIR recompilation times are significant, slowing down developers substantially. On the other hand, xDSL is removing that friction on both high and low-end hardware.

6.2 Runtime Performance of xDSL

We measure the compile-time performance of xDSL and MLIR by comparing the parsing and printing time of the MLIR test suite, as well as the runtime of a simple constant folding pass on large synthetic files. We compare the runtime performance of MLIR compiled in release and debug mode against the runtime of xDSL, both from a new Python subprocess and a tool that already preloaded all available dialects. The reason for this is that the Python decorators used in PyRDL (Section 5.2) have a fixed cost when running multiple files from the same Python script, and this cost is currently the bottleneck when running small tests, compared to MLIR, which is optimized to have a fast loading time.

Parsing and printing. We compare the parsing and printing time between xDSL and MLIR by parsing and printing all files in the MLIR test suite, as well as large synthetic files. We parse and print 96.2% of the 4868 programs in the MLIR test suite (Figure 10), where failing tests are due to our partial implementation of the `builtin` attribute parsers and printers. Overall, we observe that MLIR compiled in Debug mode, which is the standard way of using MLIR during prototyping, takes 28 seconds to parse and print all tests. When compiled in Release mode, the same benchmark takes 6 seconds. xDSL is more than an order of magnitude slower to parse the same files, and takes 266 seconds. This is due to the cost of launching the interpreter and importing xDSL for every test. If all the files are processed in the same interpreter context, xDSL takes 17 seconds, which is roughly similar to MLIR compiled in Debug mode. When parsing and printing large files (Figure 11), we observe that the same trend holds, besides that Python overhead is not significant anymore, and thus removing it does not provide any benefits. Note that xDSL does not define all dialects used in the MLIR test suite, and that some attributes and types are parsed with a default parser, which only checks for balanced sets of brackets.

Runtime. We evaluate the runtime performance of both compilers by defining a constant folding pass, which simplifies additions of constants. Most of the runtime is spent in the compiler framework code for introspecting the IR or modifying it. The pass is ran on large test cases, such that

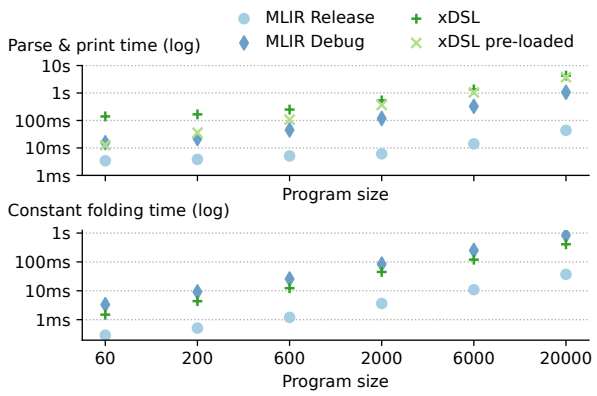

Figure 11. While parsing and printing large files in xDSL is an order of magnitude slower, the introspection and modification of IR is comparable to MLIR compiled in Debug mode.

the rewrite is applied on each possible operation, to show how the performance of xDSL scales compared to MLIR (Figure 11). Overall, the runtime of the pass is around twice as fast as the same pass in MLIR compiled in debug mode and is around an order of magnitude slower than MLIR compiled in release mode. While MLIR compiled in release mode is clearly preferable when runtime speed is a concern, xDSL has comparable speed to MLIR when compiled for development and debugging.

7 Features Enabling SideKick Compilation

While working on xDSL, we identified several features enabling the sidekick compiler approach. These features are not unique to xDSL or MLIR, and we believe that they are fundamental for building a sidekick compiler framework.

A small shared IR representation. The *key* essential feature for a sidekick compiler framework is a shared textual IR. Although the MLIR textual representation is designed to be human-readable, this is not necessary, but proves very helpful for debugging. Moreover, having a small IR greatly helps in sharing it among projects. Despite the seemingly straightforward MLIR IR, developing its parser and printer was predominantly consumed by implementing the complex syntax of builtin attributes and types, which deviate from the standard format.

Stability of core concepts and IR. Stability of the core concepts and IR of a compiler project is another essential element for enabling sidekick compilation. If the core concepts or IR are frequently changing, it is hard to stay up-to-date across projects, and incurs a significant maintenance cost.

Declarative abstraction definitions. In the case of compiler frameworks, having a declarative abstraction definition drastically simplifies the porting of dialects among projects.

8 Related Work

With sidekick frameworks, we explore new ways of making compilers and compiler frameworks from different communities interact. This section describes other approaches compilers have used to connect with compilers from other communities. We also look at other compilers implemented in Python, which shares common design goals with xDSL.

Textual interoperation with LLVM. As LLVM defines a stable textual IR, multiple tools have been developed using it. For instance, Alive [27] is a popular tool that uses LLVM IR textual format to do translation validation for LLVM. Similarly, Vellvm [40] uses the LLVM IR textual format to provide mechanized formal semantics for LLVM IR. In particular, Vellvm implements a verified version of LLVM mem2reg [41] pass that can be used at any point during a compilation pipeline through the use of LLVM IR textual format. While Vellvm connects loosely with LLVM using the textual format, this connection is not extensible, and not as automatic as in the sidekick approach we propose.

Language bindings. Popular compiler frameworks provide APIs for other languages, such as Python, showing the need to interact with compilers through a high-level scripting language. For instance, llvmlite provides Python bindings for LLVM [26], and MLIR has Python bindings [7] in its main repository, which is notably used by Nelli [24] to define an eDSL for writing MLIR programs. MLIR's Python bindings expose IR constructs like dialects, operations, regions, and attributes, but also a pass manager, enabling an entire compilation flow through Python. However, the Python bindings are still bindings to a C++ framework. So, if one intends to modify or create a new dialect, or to change the behavior of MLIR, say by adding a new rewriting infrastructure, one still has to work on the C++ code. Furthermore, to interact with MLIR dialects that are not exposed yet, such as the LLVM IR dialect, the corresponding Python bindings must be added manually. In contrast, xDSL allows extending its dialects directly in Python while also allowing to export MLIR dialects. Also, xDSL is completely written in Python, so changing the framework does not require any other language.

Lowerings to other compilers. Similar to how xDSL can be used on top of MLIR and LLVM, many compilers generate an intermediate representation and hand it to a middle-end compiler to generate machine code. For instance, Clang [21], the Rust [18] compiler, and the Julia [3] compiler uses LLVM to generate machine code, allowing them to target a wide range of architectures. CVM [30], a framework for multi-level rewriting in the database domain uses Python infrastructure to define its high-level IRs, and then generates data structures of state-of-the-art execution layers like MonetDB [16] to execute the code. While these pipelines have interactions between compilers and different languages, they lack the bidirectional capability of xDSL and MLIR, enabling seamless transitions between compilation pipeline stages.

Compilers in Python. Other compilers have been implemented in Python, though these are often only targeting Python itself. For instance, Nuitka [8], an ahead-of-time Python compiler targeting C, Numba [20], a just-in-time compiler for Python targeting LLVM IR, and Scalpel [25], a static analysis framework for Python. Recently, the PyTorch [32] community introduced `torch.fx` [34], a framework to define transformations for PyTorch kernel directly in Python. While these compilers are written in Python to make them accessible to their user communities, they are restricted to their domain and are not as extensible as xDSL and MLIR.

Extensible compilers. Other compiler frameworks have similar concepts of extensibility as the heavy linking of xDSL with MLIR. For instance, JastAdd [11], Graal IR [10], both in Java, and Delite [38], built on top of Lightweight modular staging [35] in Scala, are compilers that allow to extend the IR with new abstractions. While we could have built xDSL as a sidekick of these compilers, we decided on MLIR as the base framework due to the range of abstractions it supports, compared to the IRs of these frameworks that often lack concepts such as regions, block arguments, or attributes. On the other hand, nanopass [37], an extensible compiler framework written in Scheme, is a good example of a framework exploring a design space of quick prototyping and ease of use. Compared to our work, it does not connect to an industry-ready compiler, and cannot easily be used to prototype for an existing compiler.

9 Conclusion

We introduced the concept of a sidekick compiler framework, a compiler framework that is loosely coupled to a base framework through shared core concepts while offering a community-tailored implementation that instantiates these concepts. We implemented our idea by developing xDSL, a Python-native sidekick to MLIR and demonstrated how xDSL instantiates the concepts of SSA, regions, and multi-level rewriting. Subsequently, we showed how representing IR definitions using the IRDL dialect enables the exchange of both IR definitions and programs across compiler frameworks. Our evaluation demonstrated our fast prototyping capabilities with speedups in recompilations and low installation time, making xDSL a great tool for exploratory compiler development. We leveraged this advantage in three case studies and showed that sidekick compilers can benefit various developer communities outside the classical user base of the base compiler. We also demonstrated that by expanding the MLIR ecosystem to a different design space, we can interconnect communities such that both frameworks can form a single shared compiler ecosystem.

While xDSL is tailored to the Python ecosystem where interactive development is a focus, we believe that sidekicks can be built for other communities or needs. For instance, one could imagine a sidekick focused on parallelism or distributed computing in Rust, or a sidekick focused on providing correctness guarantees in Lean or Coq. While a change in programming language can guide the choice of a new focus, this change is not fundamentally required to explore a new design space. Similarly, sidekicks implemented in different languages but with similar technical decisions can be used to connect different programming language communities. We envision that the translation approach of xDSL, i.e. exposing data structure definitions as programs that can be ported across frameworks, will be a cornerstone of a new generation of compilers that heavily leverages sidekick frameworks.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) grants EP/W007789/1 and EP/W007940/1 and has also received funding from the European Union's Horizon EUROPE research and innovation program under grant agreement no. 101070375 (CONVOLVE). We thank the xDSL community for their useful comments and discussions along the way.

A Artifact Appendix

A.1 Abstract

This is the supporting artifact for the paper titled "xDSL: Sidekick Compilation for SSA-Based compilers" as published in CGO 2025. It contains the source code of xDSL and can be used to reproduce all results presented in the final version of this paper.

A.2 Artifact Check-List

- **Program:** xDSL compiler frameworks along with their included dialects, available in this artifact and also as open-source software. Additional scripts are available to reproduce the results of the paper. A detailed description of the artifact is documented in the README.md file included in the artifact.
- **Compilation:** Publicly available and included in this artifact: xDSL 0.23.0 [2], LLVM and MLIR 19.1.3[3]. The aforementioned LLVM and MLIR 19.1.3 are built with the LLVM Clang toolchain of the Docker container.
- **Binary:** The Linux ELF binaries of the aforementioned LLVM and MLIR compilers are included in the Docker container image.
- **Run-time environment:** The Docker container image platform is `linux/amd64`.
- **Metrics:** Time measurements for installation, (re)compilation, printing/parsing and constant folding. File size for download and installation of compiler frameworks. Number of IR features (dialect operations and attributes).
- **Output:** Figures 5 and 9 to 11 can be reproduced, as well as the values for Figure 8.

[2] commit: c6c4093bca740318870a32642856f7b5a8c75bdf
[3] commit: 98e674c9f16d677d95c67bc130e267fae331e43c

- **How much disk space required (approximately)?:** All assets require a total of 95 GB of disk space.
- **How much time is needed to complete experiments (approximately)?:** Highly dependent on CPU and clocked frequency. The execution of all experiments requires 2 hours and 30 minutes on an AMD Ryzen 9 5950X 16-core CPU at 4.9 GHz with 62 GiB RAM.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** 10.5281/zenodo.14263271.
- **Code licenses (if publicly available)?:** Apache License version 2.0 with LLVM Exceptions.

A.3 Description

A.3.1 How Delivered. The artifact [13] is available at 10.5281/zenodo.14263271.

A.3.2 Hardware Dependencies. An x86-64 machine with at least 16GB of RAM and 200 GB of free disk space is required to run the artifact. At least 8 CPU cores are recommended to reproduce the results in a reasonable amount of time.

A.3.3 Software Dependencies. Working Docker installation. This has been tested on a Linux x86 host machine with Docker 27.3.1. The artifact should work on MacOS with Apple Silicon, but it will have noticeable performance differences due to the x86 architecture simulation.

A.4 Installation

A.4.1 Experiments Repository. Download the experiments repository tarball, navigate to the directory containing the download and extract it:

```
$ tar xvfz xdsl-paper-experiments.tar.gz
```

A.4.2 Setting up the Docker Container. Download the Docker image containing the MLIR and pre-installed xDSL toolchains, navigate to the directory containing the download, load and start it:

```
$ docker load --input xdsl-toolchain-artifact.tar.gz
```

A.4.3 Building the Project. Run the container from the directory containing our experiments repository and build the project (this requires only a few minutes):

```
$ docker run -ti \
  --volume ${PWD}/xdsl-paper-experiments/:/workbench/experiments \
  xdsl-toolchain-artifact
$ cd experiments
$ make build
```

A.5 Experiment Workflow

A.5.1 Execution of all Benchmarks. In the docker container, navigate to the experiments directory (if not already there):

```
$ cd experiments/
```

Run all the experiments:

```
$ make artifact
```

Produce the figures (generated in the `plots/` directory):

```
$ make plots
```

A.6 Evaluation and Expected Result

On a similar environment, all results should closely follow the trends presented in this paper.

A.7 Notes

- Depending on the host machine configuration, `docker` commands might require elevated privileges (e.g., `sudo`).

References

[1] S. V. Adams, R. W. Ford, M. Hambley, J. M. Hobson, I. Kavčič, C. M. Maynard, T. Melvin, E. H. Müller, S. Mullerworth, A. R. Porter, M. Rezny, B. J. Shipway, and R. Wong. 2019. LFRic: Meeting the Challenges of Scalability and Performance Portability in Weather and Climate Models. *J. Parallel and Distrib. Comput.* 132 (Oct. 2019), 383–396. doi:10.1016/j.jpdc.2019.02.007

[2] S. V. Adams, R. W. Ford, M. Hambley, J. M. Hobson, I. Kavčič, C. M. Maynard, T. Melvin, E. H. Müller, S. Mullerworth, A. R. Porter, M. Rezny, B. J. Shipway, and R. Wong. 2019. LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models. *J. Parallel and Distrib. Comput.* 132 (2019), 383–396. doi:10.1016/j.jpdc.2019.02.007

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (Jan. 2017), 65–98. doi:10.1137/141000671

[4] George Bisbas, Anton Lydike, Emilien Bauer, Nick Brown, Mathieu Fehr, Lawrence Mitchell, Gabriel Rodriguez-Canal, Maurice Jamieson, Paul H. J. Kelly, Michel Steuwer, and Tobias Grosser. 2024. A shared compilation stack for distributed-memory parallelism in stencil DSLs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 38–56. doi:10.1145/3620666.3651344

[5] The CIRCT Community. 2022. "CIRCT" / Circuit IR Compilers and Tools. https://circt.llvm.org. Accessed: 2022-11-06.

[6] The LLVM Community. 2022. LLVM Loop Terminology. https://llvm.org/docs/LoopTerminology.html. Accessed: 2022-10-12.

[7] The MLIR Community. 2022. MLIR Python Bindings. https://mlir.llvm.org/docs/Bindings/Python/. Accessed: 2022-10-12.

[8] The Nuitka Community. 2022. Nuitka the Python Compiler. https://nuitka.net. Accessed: 2022-10-12.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop.*

[11] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd Extensible Java Compiler. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 773–774. doi:10.1145/1297846.1297881

[12] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. 2022. IRDL: an IR definition language for SSA compilers. 199–212. doi:10.1145/3519939.3523700

[13] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Paul Lücke, Théo Degioanni, Christos Vasiladiotis, Michel Steuwer, and Tobias Grosser. 2024. Artifact of "xDSL: Sidekick Compilation for SSA-based Compilers". Zenodo. doi:10.5281/zenodo.14263271

[14] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias

Grosser. 2021. Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation. *ACM Transactions on Architecture and Code Optimization* 18, 4 (Sept. 2021), 51:1–51:23. doi:10.1145/3469030

[15] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving High-Performance the Functional Way: A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (aug 2020), 29 pages. doi:10.1145/3408974

[16] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35 (01 2012).

[17] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2389–2401. doi:10.14778/3551793.3551801

[18] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language (Covers Rust 2018).* No Starch Press.

[19] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. *Jupyter Notebooks-a publishing format for reproducible computational workflows.* Vol. 2016.

[20] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.

[21] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004. (CGO '04).* IEEE Computer Society, Washington, DC, USA, 75–. doi:10.1109/CGO.2004.1281665

[23] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[24] Maksim Levental, Alok Kamatar, Ryan Chard, Nicolas Vasilache, Kyle Chard, and Ian Foster. 2023. nelli: a lightweight frontend for MLIR. arXiv:2307.16080 [cs.PL]

[25] Li Li, Jiawei Wang, and Haowei Quan. 2022. Scalpel: The Python Static Analysis Framework. arXiv:2202.11840 [cs.SE]

[26] The llvmlite Community. 2022. llvmlite Python Bindings. https://github.com/numba/llvmlite. Accessed: 2022-11-06.

[27] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/2737924.2737965

[28] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. 2020. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.* 46, 1, Article 6 (apr 2020), 28 pages. doi:10.1145/3374916

[29] Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. 2020. The Collection Virtual Machine: An Abstraction for Multi-Frontend Multi-Backend Data Analysis. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. doi:10.1145/3399666.3399911

[30] Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. 2020. The collection virtual machine: an abstraction for multi-frontend multi-backend data analysis. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 1–10.

[31] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 41–45. doi:10.1145/3358711.3361627

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[33] Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. 2022. QSSA: An SSA-Based IR for Quantum Computing. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC)*. Association for Computing Machinery, New York, NY, USA, 2–14. doi:10.1145/3497776.3517772

[34] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch. fx: Practical Program Capture and Transformation for Deep Learning in Python. *Proceedings of Machine Learning and Systems* 4 (2022), 638–651.

[35] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 127–136.

[36] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, 12–27. doi:10.1145/73560.73562

[37] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2005. EDUCATIONAL PEARL: A Nanopass Framework for Compiler Education. *J. Funct. Program.* 15, 5 (Sept. 2005), 653–667. doi:10.1017/S0956796805005605

[38] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–25.

[39] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. *CoRR* abs/2202.03293 (2022). arXiv:2202.03293 https://arxiv.org/abs/2202.03293

[40] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.* 47, 1 (jan 2012), 427–440. doi:10.1145/2103621.2103709

[41] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM. *SIGPLAN Not.* 48, 6 (June 2013), 175–186. doi:10.1145/2499370.2462164

Received 2024-09-12; accepted 2024-11-04