Performance and Dynamism in User-extensible Compiler Infrastructures

Edmund Goodman
Clare College

UNIVERSITY OF CAMBRIDGE

June 11, 2025

Submitted in partial fulfilment of the requirements for the Master of Philosophy in Advanced Computer Science

Total page count: 118

Main chapters (excluding front-matter, references and appendix): 53 pages (pp 13–65)

Main chapters word count: 14933

```
$ texcount -inc -total -sum -brief report-submission.tex
14933
```

Listing: Methodology used to generate that word count.

Declaration

I, Edmund Goodman of Clare College, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Signed: Edmund Goodman

Date: June 11, 2025

Abstract

Performance and Dynamism in User-extensible Compiler Infrastructures

MLIR is a modular compiler framework that provides core infrastructure to be leveraged and extended by users implementing their own compilers, an inherently dynamic design as a result of the underlying heterogeneous data structure whose shape is known only at runtime. This approach presents an inherent optimization boundary, as the dynamic structures cannot be precisely reasoned about before runtime to guarantee the validity of optimisations, meaning static ahead-of-time compilation provides fewer benefits. Previous compiler frameworks accept the limitations of this optimization boundary, leveraging only the remaining optimizations offered by static ahead-of-time compilation, yet still incurring the costs of long build times and reduced flexibility, suggesting dynamic languages might be more suitable. We examine performance bottlenecks incurred by dynamic languages for code rewriting tasks in xDSL, a Python-native compiler framework inspired by MLIR. We find that both the inherent dynamism of these rewriting tasks over runtime heterogeneous data structures and modern interpreter optimisations narrow the performance gap between static and dynamic languages, using both traditional measurement techniques and a novel tool for performance profiling bytecode instructions. Our research challenges the status quo of implementing user-extensible compiler frameworks in static, ahead-of-time compiled languages. Instead, we motivate the use of dynamic languages, demonstrating that they balance compilation performance with the flexibility and fast build times.

Acknowledgements

This project would not have been possible without the wonderful support of my co-supervisor Sasha Lopoukhine, whose insight and advice has been indispensable, and my supervisor Dr Tobias Grosser, whose project ideas and guidance throughout the project have been invaluable. I'd also like to thank the rest of the Grosser Lab, in particular Mathieu for his xDSL and MLIR expertise, along with Ivan, Emma, and Léo for their insights and friendship in the office. Finally, I'd like to thank my parents, Jonathan and Victoria, for their perpetual support and well-reasoned advice, especially in times when life is difficult.

Contents

List of Abbreviations

FFI Foreign Function Interface. 37, 60

GCC GNU Compiler Collection. 17

GEMM General Matrix Multiply. 18, 56

GIL Global Interpreter Lock. 34

IR Intermediate Representation. 9, 13, 15, 17, 18, 22–27, 43, 45, 48, 55, 59, 63

IRDL Intermediate Representation Definition Language. 18, 39

JIT just-in-time. 12, 19, 20, 31, 50–52, 59, 63

MLIR Multi-Level Intermediate Representation. 9, 10, 12–14, 17, 18, 21–29, 38–47, 53, 55–57, 59, 60, 63, 64

PEP Python Enhancement Proposal. 48, 50

PyPI Python Package Index. 31, 32, 37

RISC Reduced Instruction Set Computer. 17, 55

RTTI Run-time Type Information. 10, 42, 56

SSA Static Single Assignment. 13, 17, 18

VM Virtual Machine. 48

vtable Virtual Method Table. 53–55, 61

List of Figures

List of Tables

Chapter 1

Introduction

Compilers are a critical component of computing systems, providing an abstraction from high-level programming languages to the underlying machine ISA. Released in 1987, `gcc` was a popular open-source compiler [1], but its monolithic and tightly-coupled design impacted its re-usability. The LLVM compiler framework [2] addressed these problems with a human-readable and language-independent textual IR in Static Single Assignment (SSA) form [3], which could be analysed and transformed by a sequence of passes, made available as a re-usable C++ library. Recently, MLIR [4], has furthered these goals by enabling users to cheaply extend compilers with their own abstractions.

Compiler extensibility is critical for handling the heterogeneous hardware and exotic optimisations of modern workloads. To implement this, MLIR uses dynamic, pointer-chasing data structures and algorithms to represent its IR. For example, pattern rewriting in MLIR traverses and pattern matches on a linked list representation of SSA values. This further inhibits optimisations, both as a result of dynamism and not being amenable to other transformations such as vectorisation. These factors motivate challenging the status quo of LLVM and MLIR implementing user-extensible compiler frameworks in static, ahead-of-time compiled languages.

xDSL [5] is a reimplementation of MLIR's core data structures and IR definitions in Python, a dynamically typed, interpreted language. Using Python allows xDSL to take MLIR's goal of user extensibility to even further extremes by leveraging its dynamic features such as runtime meta-programming. In addition to this, xDSL's interpreted nature avoids MLIR's long build times, and its dynamic typing matching the dynamic nature of the user-extensible compiler framework workload. However, using Python also has drawbacks, most notably in relation to runtime performance. This work examines the performance of the pattern rewriting component of the xDSL compiler framework, focussing on the impact of dynamism and contrasting against the current state-of-the-art, MLIR.

The performance of a program is constrained both by the details of its implementation and the runtime of its language. These two properties are deeply interlinked, making them difficult to measure independently. To disentangle them, we manually optimise and specialise xDSL's implementation of pattern rewriting (Figure 1.1, 1), resulting in an 7× performance uplift. We then confirm that the performance of the specialisation is constrained only by the language runtime by examination of the dispatched bytecode of micro-benchmarks. This bytecode examination process revealed a gap in the provision of fine grained performance profilers for Python. To address this gap, we developed ByteSight, native tracing performance profiler for Python bytecode. We then use this tool to quantify the impact of recent performance enhancements made to CPython for this specialised implementation (Figure 1.1, 2). This describes the best-case for the performance of pattern rewriting in xDSL, which can then be compared against MLIR.

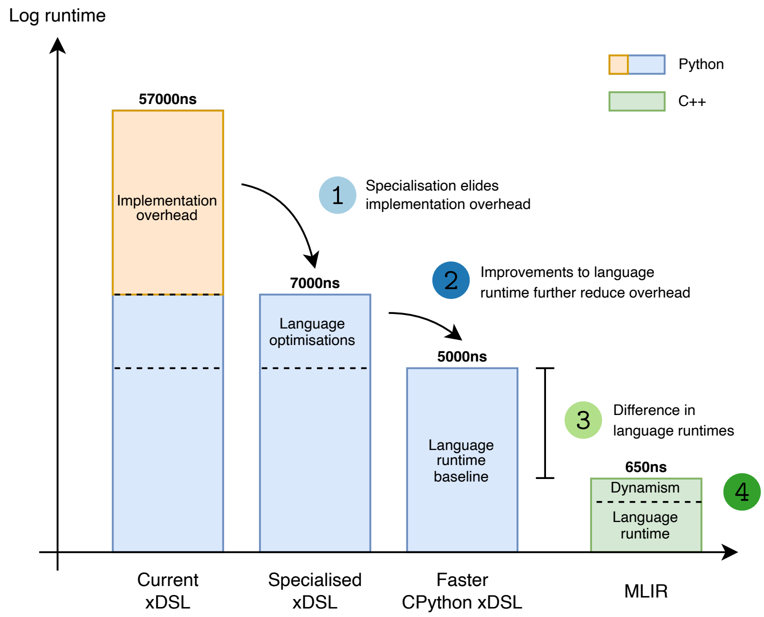


Figure 1.1: Closing the gap between xDSL and MLIR pattern rewriting performance. xDSL's performance can be improved by changes to both its implementation and language runtime. C++ has less of a performance advantage for MLIR than other workloads due to the costs associated with dynamism.

A key difference between the Python and C++ runtimes is their degree of language dynamism. MLIR's C++ runtime incurs overhead when dynamically dispatching functions (Figure 1.1, 3), which is worsened by prohibiting ahead-of-time performance optimisations. In contrast, almost every bytecode operation evaluated by the Python interpreter is dynamic, each incurring an overhead checking runtime information. As such, we expect the difference in performance between language runtimes (Figure 1.1, 4) to be smaller for more dynamic workloads. To corroborate this, we measure the difference in performance between pattern rewriting workloads using xDSL and MLIR, and assess the contribution of overheads incurred by dynamism. This measurement procedure uniquely leverages xDSL's sidekick compilation functionality to ensure the comparability of performance measurements by driving them with the same textual IR, even for implementation details internal to each framework. Finally, we critically evaluate the degree to which this motivates the use of Python for implementing user-extensible compiler frameworks.

The contributions of our work are as follows:

- An examination of the current performance of pattern rewriting workloads in the CPython language runtime (chapter 3).
- A tool to examine CPython bytecode dispatch in program runs, facilitating the analysis of costs incurred by dynamism (chapter 4).
- A specialisation of the current xDSL implementation for pattern rewriting workloads, demonstrating the best-case performance of CPython for such applications (chapter 5).
- An exploration of optimisation techniques to shrink the performance gap between dynamic and static languages for pattern rewriting workloads (chapter 6).
- A quantitative comparison of the performance of user-extensible compiler frameworks implemented in static and dynamic languages, focussing on the impact of dynamism (chapter 7).

Chapter 2

Background

Compilers are the interface between application code and the underlying hardware on which it is executed. With the end of Dennard scaling and the slow-down of Moore's law [6], the design of computer hardware increasingly relies on heterogeneous accelerator hardware to achieve performance goals within power and area budgets [7]. This has been accompanied by a stratospheric increase in the amount of compute being used for the training and inference of large machine learning models [8]. Together, these two factors make the development of compilers which can perform high-level optimisations and target exotic accelerator hardware critical for delivering the performance goals of modern applications. Since both the optimisations and underlying hardware are fast-changing, the development velocity of the compiler is crucial to deliver state-of-the-art models with delay (Figure 2.1) [9]. This motivates the development of shared compiler infrastructure which increases development velocity, for example by having short build times, a simple syntax, and a convenient API.

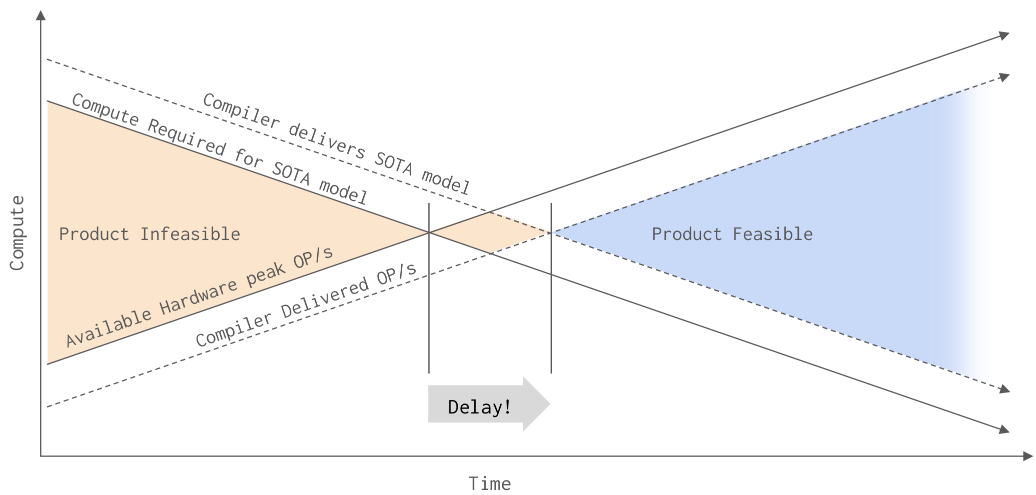


Figure 2.1: Compilers are lagging behind the development of both machine learning hardware and workloads. Figure created by Anton Lydike based on a slide from Sean Silva's talk at the March 2025 Cambridge Compiler Social.

Early compilers such as Grace Hopper's A0 and the ALGOL compiler were standalone programs, specific to individual languages [10]. Researchers then generalised these standalone programs into shared infrastructure, for example with the SUIF project [11]. The GNU Compiler Collection (GCC) project adopted these ideas, providing a popular and open-source shared infrastructure for the compilers of several languages, including C, C++, and FORTRAN. However, its monolithic and tightly-coupled design limited the reusability of its components [1], presenting a research gap in the field.

2.1 The LLVM project

In 2004, Lattner and Adve published "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation" [2], productising a collection of good ideas in compiler design. One of LLVM's key contributions was a human-readable textual IR, contrasting the binary IRs of `gcc` at the time. This IR has a language- and target-independent RISC-like instruction set in SSA form [3], making LLVM re-usable across different frontends and backends, while benefitting from shared infrastructure. Modular transformation passes can be defined to modify this IR. These passes can be composed to implement sophisticated optimisation pipelines, as each pass operates on the same IR structures. These desirable properties lead to LLVM's widespread adoption, but the demands of modern workloads present a number of challenges for LLVM. For example, LLVM's low-level IR is not well suited to abstract over the high-level operation rewrites required to optimise machine learning workloads. In addition to this key infrastructure such as parsing and printing must be implemented by hand, slowing development velocity.

2.2 Multi-level intermediate representation (MLIR)

Multi-Level Intermediate Representation (MLIR) addresses these modern challenges, providing "compiler infrastructure for the end of Moore's law" [4]. Its key contribution is allowing users to easily define multiple levels of abstraction supporting heterogeneous recursively nested structures, referred to as dialects. This contrasts the fixed low-level abstractions and homogeneous flat structures of LLVM IR, whose shape is known at runtime, resulting in less dynamic dispatch and control flow. As a result of this, MLIR allows different problem domains to define their own operations and transformation passes while sharing common infrastructure. This facilitates efficiently expressing optimisations at the most appropriate abstraction level, which can then each be applied, lowering from high-level domain-specific representations to low-level hardware-specific ones. Lowering leverages pattern rewriting, a declarative way to express transformations as patterns which match IR constructs and replace them with lowered versions. Traits abstract across properties shared by IR components [12], facilitating these user-extensible pattern rewrites on custom dialects. The provision of common infrastructure including documentation generation, parsing and printing logic, and location tracking further reduces the engineering effort required to develop complex optimisations and support custom hardware targets.

MLIR is implemented in C++, providing desirable performance characteristics, but having a verbose syntax and long build times. This is partially mitigated by Intermediate Representation Definition Language (IRDL), "a domain-specific language to define IRs" [13], implemented as a SSA-based dialect of MLIR. IRDL provides a declarative system for defining dialects, operations, and types which can be dynamically loaded at runtime.

2.3 xDSL

xDSL is a Python-native compiler framework, originally written as a re-implementation of MLIR's data structures and associated logic. As a result of its similar data structures and shared textual IR, xDSL can be used as a side-kick compiler for MLIR, facilitating modular replacement of individual components of its pipeline. This is achieved with a DSL implementation in Python of IRDL. In contrast to MLIR, xDSL prioritises developer productivity through Python's easy-to-use syntax and fast build times, along with first-class support for user-extensibility which allows adding new operations at runtime. This design choice has benefits for modern compiler developers, where improvements to developer productivity are critical to minimise delay for state-of-the-art workloads (Figure 2.1).

2.4 Dynamic languages and workloads

Programming language design is a game of trade-offs, with a wide variety of design choices incurring differing benefits and costs. One such choice is the degree of dynamism, defined by Williams et al. as "allowing properties of programs to be defined at run-time" [14]. As such, static languages fix properties ahead of time, whereas dynamic languages offer more flexibility at runtime. These trade-offs for features including dynamic dispatch and runtime meta-programming are discussed in the literature (Related work, section 8.3).

In addition to considering support for dynamism as a property of a programming language, it can be helpful to classify a workload as dynamic or static. For example, General Matrix Multiply (GEMM) operations which underpin modern machine learning systems rely on streaming data in a statically known order. This is well-suited to ahead-of-time compilation, as it is amenable to optimisation passes requiring no runtime information, such as code motion or vectorisation [15]. In contrast, pattern rewriting in user-extensible compiler frameworks relies on pointer chasing data structures with a high degree of dynamism. The SSA representation of the code being rewritten is structured as a doubly linked list, with the applications of the rewriting semantics to this list known only dynamically at runtime. This dynamic, pointer-chasing workload is inherently memory bound, having irregular access patterns with high latencies [16]. As such, optimisations to computational throughput have minimal impact on overall performance, as the bandwidth ceiling upper bounds performance [17]. Furthermore, it precludes many optimisations leveraged by ahead-of-time statically compiled languages such as C++ to accelerate their performance for other workloads.

2.5 CPython internals

Python is a high-level, dynamically typed-checked programming language, known for its readable syntax and extensive library support [18]. The language has many implementations, from its reference implementation, CPython, to alternatives such as PyPy's JIT compiler [19]. CPython operates by compiling Python source code into a stack-based bytecode. This bytecode is executed by a virtual machine in an interpreter loop, fetching and evaluating the compiled sequence of instructions one at a time. This approach limits performance compared with static ahead-of-time compiled languages by the overhead of the virtual machine and having fewer opportunities for ahead-of-time optimisation. However, it also provides great flexibility, supporting Python's dynamic type system and runtime meta-programming capabilities. The Faster CPython project is an ongoing effort to improve the performance of Python by applying techniques such as adaptive optimisation and JIT compilation. For such efforts, it is critical to have a suite of programs with which to benchmark performance. In Python, this is provided by PyPerformance, "an authoritative source of benchmarks for all Python implementations" [20].

2.6 JIT compilation

In their paper "A Brief History of Just-In-Time", Aycock defines just-in-time (JIT) compilation as "translation that occurs after a program begins execution" [21]. He argues that JIT compilation approaches aim to accrue the benefits of both ahead-of-time compilation and interpretation, combining the performance traditionally associated with compilation with the portability and access to runtime information of interpretation.

2.6.1 JIT compilation to machine code

While just-in-time (JIT) compilation can refer to any program translation occurring at runtime, it often refers specifically to the dynamic generation of machine code just before execution. Compilation during runtime exposes information from actual program behaviour which is not available to traditional ahead-of-time compilers. This allows JIT compilers to optimise the generated machine code, and unblocks the optimisation of dynamic runtimes for which there is insufficient information to optimise ahead-of-time.

A major bottleneck for traditional JIT compilation is the speed of compiler optimisation passes, which can lead to delays during program execution. In their paper "Copy-and-Patch

Compilation", Xu and Kjolstad present a novel approach to avoid this runtime cost [22], while still generating high quality machine code. Instead of compiling code from scratch at runtime, a library of machine code stencils for common operations is constructed ahead of time, with holes left for runtime-specific information such as memory addresses. During program execution, the JIT then "patches" these holes to generate executable machine code without incurring the runtime cost of traditional optimising compilers such as LLVM.

2.6.2 Adaptive optimisation

In addition to generating machine code on-the-fly, runtime information can be used to adapt program execution to the current workload. A significant overhead in the performance of dynamically typed languages is checking types at runtime, which is required to select the matching operation implementation for the type. While this type information cannot be known ahead of time in dynamically typed languages, information collected at runtime can be used to optimised the type-checking process. This adaptive optimisation approach relies on the assumption that if a variable has had a fixed type for a sufficient span of time previously, it is likely that it will have the same type in future. For example, while a variable being used as an integer counter in a loop can take any value, if its type in the earlier iterations has always been an integer, it is likely to remain an integer throughout later iterations. In an interpreted language, this assumption can be leveraged to replace general instructions with faster, specialised variants. A concrete implementation of this is Python's specialising adaptive interpreter, discussed in section 6.1.

Chapter 3

Compiler framework performance

In this chapter, we measure the runtime performance of two user-extensible compiler frameworks: MLIR and xDSL. The usefulness of these measurements are three-fold. First, we provide insight into the performance of the current versions MLIR and xDSL in relation to each other. Second, we identify the components of xDSL's implementation which contribute most to its overall runtime, and as such are good targets for optimisation (chapter 5). Finally, we isolate small, self-contained components of the implementation which are comparable between MLIR and xDSL, through which the impact of dynamism can be examined (chapter 7).

3.1 Methodology

Accurate performance measurement is a fundamental yet notoriously fickle discipline in systems research. In this section, we discuss our experimental methodology, including: the hardware and software setup; the selection of workloads examined; and additional infrastructure constructed. Through this, we aim to justify our experimental procedure and facilitate reproducibility of results.

3.1.1 Experimental setup

We measured all experimental results for this work with the same experimental setup: an AWS EC2 virtual machine (Table 3.1a). This choice benefits experimental replicability, making it easy for future researchers to provision their own AWS EC2 instance with a similar machine configuration, and hence performance characteristics. However, there are also drawbacks associated with using virtualised cloud infrastructure for performance measurements. Unlike bare-metal machines, the added layer of indirection of the hypervisor adds noise and confounding effects to the performance measurements. We use the subset of machine configuration available through the hypervisor to minimise noise, for example by pinning workloads to individual virtual cores and disabling address space randomisation.

Following these mitigations, the experimental setup is a fair compromise of leveraging available resources and empowering reproducibility for slightly reduced experimental precision. Finally, the ubiquitous use of cloud resources, especially for compilation workloads in build servers, makes this choice and its associated properties representative of real-world applications.

Table 3.1: Summary of the experimental setup used for performance measurement.

(a) Hardware configuration.

Configurable	**Configuration**
AWS instance type	c5a.4xlarge EC2
Operating system	Ubuntu 24.04
Linux kernel version	6.8.0-1029-aws
CPU name	AMD EPYC 7R32
Logical CPU cores	16
Clock frequency	2799.99 MHz
L1 Data Cache	32 KiB
L1 Instruction Cache	32 KiB
L2 Unified Cache	512 KiB
L3 Unified Cache	16384 KiB
RAM [GB]	16

(b) Software configuration.

Configurable	**Configuration**
CMake version	3.28.3
Ninja version	1.11.1
Python interpreter	CPython 3.10.17
C++ compiler	clang 18.1.8
xDSL commit SHA	`e1b12a2`
MLIR commit SHA	`6516ae4`

In addition to the underlying experimental hardware, the software configuration of the machine significantly contributes to its performance characteristics. As such, we similarly provide a description of the language versions and build tools, along with the exact `git` SHAs of the compilation frameworks that we compare in this chapter (Table 3.1b). Experiments in later chapters ablate across both language and framework versions, with these versions being noted explicitly when changed.

3.1.2 Experimental workloads

One novel contribution of xDSL is the concept of sidekick compilation frameworks: "an approach that uses multiple frameworks that interoperate with each other by leveraging textual interchange formats [...]" [5]. We leverage xDSL's interoperability with MLIR's textual IR to construct workloads which can be applied to both frameworks. This guarantees directly comparable performance measurements at the pipeline phase and function level, driven by the shared representation. Without sidekick compilation, for example comparing two more disparate user-extensible compiler frameworks such as MLIR and Pliron [23], this comparison would be constrained to only end-to-end measurement, obscuring the impact of variables such as dynamism on the performance properties.

MLIR facilitates the representation of algorithms of varying complexity across a wide range of abstraction levels. In this case, we define workloads as the combination of a segment of IR and a rewriting optimisation transforms it. This can be used to drive both frameworks' command line optimiser tools and internal phases and functions within the frameworks. Although real-world workloads vary in structure, a common use-case for MLIR is ingesting IR, applying rewriting operations to it, and finally passing the transformed IR to LLVM for code generation. Since this IR is often provided in a binary format in production environments, as opposed to a textual one more commonly used during development, rewriting machinery constitutes a high proportion of the executed logic. Because of this, workloads representative of these real-world use-cases should exercise common operations in rewriting machinery, such as walking through operations, checking their traits, and replacing them in their parent block. In addition to this, using basic dialects through which most IR are lowered such as `arith` and `builtin` is similarly beneficial. As such, we select constant folding as our main experimental workload to fulfil these requirements.

Constant folding

Stoltz et al. define constant folding as "a well-known static compiler technique in which values of variables which are determined to be constants can be passed to expressions which use these constants" [24]. It has been leveraged for both performance and space improvements since the earliest optimising compilers. Despite this, it remains relevant even over more modern, exotic optimisations due to its simplicity, efficient implementation, and high impact on generated IR. In addition to this, procedural generation of IR which can be constant folded is simply parameterised in length (section B.1), facilitating experiments examining performance scaling. We show an example IR segment with three constants below (Listing 3.1).

```
builtin.module {
  %0 = arith.constant 865 : i32
  %1 = arith.constant 395 : i32
  %2 = arith.addi %1, %0 : i32
  %3 = arith.constant 777 : i32
  %4 = arith.addi %3, %2 : i32
  "test.op"(%4) : (i32) -> ()
}
```

(a) Unfolded IR amenable to constant folding.

```
builtin.module {
  %0 = arith.constant 2037 : i32
  "test.op"(%0) : (i32) -> ()
}
```

(b) Constant folded IR.

Listing 3.1: Folding three integer constants over arithmetic addition in MLIR's textual IR.

3.1.3 Measurement infrastructure

In order to facilitate the efficient and reproducible measurement of xDSL's performance characteristics, we developed infrastructure to drive our benchmarks with a variety of measurement tools and profilers. In addition to their usefulness for understanding and optimising compiler performance, the benchmarks composing the performance experiments also provide an opportunity to augment the development process of the xDSL project. Benchmarks can be used to characterise the performance impact of changes to the xDSL codebase, making it easier to avoid unnecessary performance regression. As such, we provide a command line interface for developers to run the benchmarks, with further functionality which supports a variety of profiling tools[1]. Furthermore, our benchmarks interface with Airspeed Velocity [25], a tool which runs benchmarks across repository commits. The xDSL website then tracks the results of this tool, providing a dashboard for the performance characteristics of xDSL over time[2].

3.2 End-to-end benchmarks

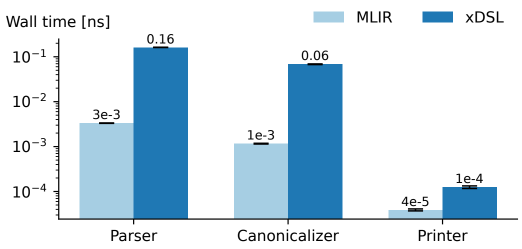
The simplest metric for the performance of a system is its overall runtime. Both xDSL and MLIR provide two APIs in the form of a command line tools and a programmatic interface. Although real-world workloads typically pass binary IR to a subset of the programmatic interface, this does not fully exercise the overall functionality of the system. In contrast, measuring the end-to-end runtime of the command line optimiser for textual IR captures a large proportion of the functionality, including debugging tools such as parsing and printing the textual IR. This makes them a good candidate for this simple metric of end-to-end runtime.

End-to-end measurements are well-aligned metrics of performance for real-world workloads, but are limited by their coarse granularity. Conveniently, the structure of modern compiler frameworks facilitates more detailed breakdowns, even for end-to-end runs. LLVM's key contribution of a human-readable textual IR facilitates splitting compilation into a sequence of discrete re-usable phases, which can be measured independently. As such, compilers having LLVM's pedigree, such as MLIR and xDSL, can be modelled as a pipeline – parsing the input, applying optimisation transformations, and generating a lowered output. This allows us to break down end-to-end benchmarks into components of finer granularity.

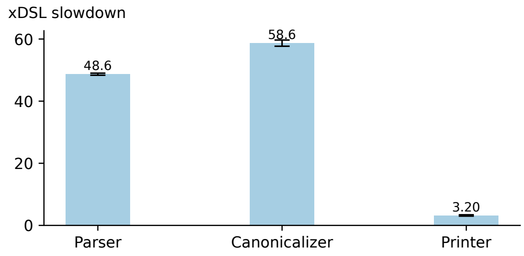
For MLIR, we invoke `mlir-opt constant_folding.mlir --canonicalize --mlir-timing`. By default, this prints results to four decimal places, which is insufficiently granular to time printing very short IR segments. As such, we modify MLIR's implementation to print more decimal places. In order to guarantee statistical significance following this change, we measure across ten repeats. Since we are sampling a single variable, we take

[1] `https://github.com/xdslproject/xdsl/blob/main/benchmarks/README.md`
[2] `https://xdsl.dev/xdsl-bench/`

the arithmetic mean of these values, with an uncertainty given by their standard deviation. For xDSL, we leverage our custom benchmarking infrastructure (subsection 3.1.3), which drives each phase canonicalizing the workload and records precise timings. We then calculate the average and uncertainty by the same procedure, in order to plot these wall times for each framework and the slowdown between them (Figure 3.2a).



(a) All pipeline phases of MLIR are faster than xDSL.



(b) xDSL's parser and canonicalizer incur around 50× slowdown, whereas the printer slows down only 2× for small workloads.

Figure 3.2: Performance comparison of constant folding 1000 integer addition operations (section 3.1.2) between xDSL and MLIR.

Despite its simplicity, this initial experiment reveals a number of interesting insights. Firstly, the slow-down from xDSL to MLIR for the first two passes is approximately 50×. This provides an initial baseline for the overhead incurred by the combination of xDSL's language runtime and implementation details over the MLIR implementation. Secondly, the slow-down for the printer is much lower than the other two phases, at only 3×. The constant folded output is very short, containing only two IR instructions (Listing 3.1b) for any length input of that schema. As such, the overhead of writing to the standard output is greater than the processing time for those few operations. In contrast, the logic implement parsing and canonicalization rewrites is much longer than any fixed overhead they incur, such as the parser reading the IR from a file.

Since the parser represents a majority of the total runtime for both frameworks, it appears an attractive candidate for optimisation as a corollary of Amdahl's law [26]. Of any pipeline phase, improving the parser by a fixed proportion yields the greatest runtime reduction. Despite this, we select the pattern rewriting phase, which implements the canonicalizer, for further investigation. Compilers designed to meet the demands of modern workloads heavily focus on optimising IR through rewriting. For example, the XLA machine learning compiler takes a binary input representation as input, which it then lowers through MLIR dialects applying rewrite optimisations in order to generate efficient code targeting accelerator hardware [27]. This process only relies on parsing and printing logic for debugging, as its main code generation flow operates entirely on binary representations. As such, optimising these components is a less attractive research direction, as it would only affect the performance of development and testing infrastructure. Furthermore, the rewriter is a much more dynamic workload, with its control flow necessarily determined at runtime based on the contents of the IR being optimised. This further makes it a more suitable candidate for examination through the lens of the impact of dynamism on user-extensible compiler frameworks.

3.3 Micro-benchmarks

Micro-benchmarking refers measuring the performance of fast, granular, and isolated segments of code. The term was coined by Saavedra et al. in their 1995 paper [28] "Performance Characterization of Optimizing Compilers". As such, we are in good company in our application of micro-benchmarking approaches to this problem domain. Micro-benchmarks have many desirable properties. Since they run quickly, they can cheaply be repeated for statistical confidence. Furthermore, their fine granularity makes them tractable to reason about – providing useful information to optimise the component of the system they measure. However, a key difficulty of micro-benchmarking is ensuring alignment with overall system performance. For example, the selection of code paths to micro-benchmark may introduce bias, making them less representative of the overall system. In addition to this, their performance may be inflated as a consequence of warmed caches and JIT optimisations across repeats, which would not occur during normal operation [29]. As such, micro-benchmarking is a useful tool for deeply understanding the performance of software, but must be used carefully to ensure the validity of its results.

As discussed in our related work (Section 8.1), Amini and Nui's talk "How Slow is MLIR" [30] discusses a set of micro-benchmarks for key operations in the MLIR compiler, such as traversing the IR and creating operations. These micro-benchmarks were used to inform the optimisation of MLIR's data structures and for comparison with traditional LLVM-based compilers. The implementation of the micro-benchmarks allude to an underlying design goal in MLIR by their measurement of asymptotic scaling properties[3]. This design goal is asymptotically optimal performance for its underlying data structures. However, data structures with these characteristics often incur constant-time penalties. This causes overhead for small workloads, where, unlike the asymptotic case, the cost is not amortised. As such, micro-benchmarks may not be representative of the system's overall performance, revealing possibility for the optimisation of code co-designed using them. Despite this, they can still provide useful insight into MLIR's performance characteristics. We implement micro-benchmark workloads for the xDSL equivalent to those for MLIR presented in the keynote, and compare the results of these benchmarks between the two implementations, giving insights into their relative performance.

[3] `https://github.com/joker-eph/llvm-project/blob/6773f8/mlir/unittests/Benchmarks/Cloning.cpp#L66`

3.3.1 Implementation

Unfortunately, the implementation and build instructions for the "How Slow is MLIR?" micro-benchmarks were not published with the talk. However, the source code can be found on a branch of their fork of LLVM[4]. We provide a copy of this source code and instructions for running the benchmarks[5]to enhance the replicability of our results and facilitate further performance experiments. We then use this source code to construct comparable micro-benchmarks in Python.

A key design goal of our micro-benchmarks for xDSL is parity with those provided for MLIR, ensuring the validity their direct comparison. As such, we derive their implementation from the MLIR benchmarks, matching test data and function invocations as closely as possible. In addition to being fairly comparable, it is critical that the micro-benchmark results are statistically significant. This is particularly challenging as their execution time approaches the 1ns resolution of the most granular system counters on the benchmarking machine. As such, we measure the total time taken to execute the functions under test 32768 times, evaluating the individual duration by dividing by the number of iterations. As with the traditional benchmarks, we repeat this process ten times, calculating the arithmetic mean and standard deviation as the value and its uncertainty respectively. In the following sections, we discuss the implementations of a number of micro-benchmarks, facilitating discussion of the insights they give into compiler performance across implementations and language runtimes in later chapters.

3.3.2 Operation instantiation

Operations are a central data structure in MLIR and xDSL's IR representation, constituting dialects and composing together into programs. As such, the methods to instantiate operations a very frequently invoked, making them a good candidate for micro-benchmarking. In both MLIR and xDSL, there are two main ways an operation can be constructed: direct creation 1; or using a builder 2 (Listing 3.3).

[4] https://github.com/joker-eph/llvm-project/tree/benchmarks
[5] https://github.com/EdmundGoodman/llvm-project-benchmarks

```
// Setup
OpBuilder b(ctx.get());

// Benchmark 1
OperationState opState(unknownLoc,
    "testbench.empty");

Operation::create(opState);
// Benchmark 2
b.create<EmptyOp>(unknownLoc);
```

(a) "How Slow is MLIR?" C++ implementation.

```
# Benchmark 1
EmptyOp.create()

# Benchmark 2
EmptyOp()
```

(b) xDSL Python implementation.

Listing 3.3: Micro-benchmark implementations for methods checking an operation has a trait.

The performance characteristics of these implementations differ significantly (Table 3.2). In MLIR, the creation and building mechanisms have approximately the same performance, taking around 150ns to instantiate an operation. In contrast, xDSL's operation building is much slower than direct creation, jumping from 20× to 80× slower. This comes as a result of the trade-off xDSL makes between performance and expressivity, with its builder including a large section of inference and verification logic as a wrapper around the create method. While MLIR disables this verification logic for release builds, it is always enabled in xDSL, facilitating its use case of development and prototyping. We examine the performance impact of disabling verification in the later discussion of workload specialisation (chapter 5).

Table 3.2: Operation instantiation in xDSL is approximately 25× slower than in MLIR when creating instructions in the asymptotic case, but is up to 90× slower when using the builder API.

Mechanism	MLIR [ns]	xDSL [ns]
1 Create	155 ± 14	3770 ± 916
2 Build	153 ± 0.5	12700 ± 1810

3.3.3 Operation trait checks

Traits are a key property of MLIR and xDSL's operations, providing a mechanism to abstract implementation details and properties. In order to allow users of the frameworks to leverage an operation's trait information, both MLIR and xDSL provide helper methods to check whether an operation has a certain trait. These methods are used frequently in common tasks. For example, when pattern rewriting over a block of IR, the matching engine uses the traits of the block's constituent operations to identify valid rewrites.

```cpp
// Setup
Operation op = b.create<OpWithRegion>(
    unknownLoc
);

// Benchmark
bool hasTrait = op->hasTrait<
    OpTrait::SingleBlock
>();
```

(a) "How Slow is MLIR?" C++ implementation.

```python
# Setup
op = OpWithRegion()

# Benchmark
has_trait = op.has_trait(SingleBlock)
```

(b) xDSL Python implementation.

Listing 3.4: Micro-benchmark implementations for methods checking an operation has a trait.

These methods can be simply benchmarked for an example workload (Listing 3.4). However, care must be taken when implementing these benchmarks to ensure they are can be equitably compared. For example, the operations must both have the same number of traits, and the target trait must be at the same position in that operation's internal data structure. If this is not true, one implementation may require fewer iterations to perform its workload, invalidating the comparability of the two results. Micro-benchmarks of checking traits for both implementations show a slow-down of approximately 250× from MLIR to xDSL (Table 3.3).

Table 3.3: Trait checks in xDSL are approximately 250× slower than in MLIR in the asymptotic case.

MLIR [ns]	xDSL [ns]
11.7 ± 0.5	1920 ± 695

3.3.4 Summary of micro-benchmarks

These micro-benchmarks provide a number of insights into the baseline performance of xDSL. Firstly, we can see that xDSL's implementation incurs a performance trade-off for the expressivity of its API, with convenient helper methods performing much less well than direct interactions with the underlying data structures. This motivates our work specialising xDSL for a single workload to elide this overhead (chapter 5). Secondly, we begin to quantify the performance overhead of the CPython language runtime over C++ through workloads with trivial implementations such as instantiating empty operations. From this, we can see that CPython 3.10 incurs at least a 25× slowdown over C++. This motivates our work understanding the impact of recent optimisations to CPython such as the specialising adaptive interpreter and baseline JIT (chapter 6), along with the necessary cost of dynamism required to implement Python's semantics (chapter 7).

In addition to the micro-benchmarks matching those provided in “How Slow is MLIR”, a subset of which is discussed in detail above, we also developed a further xDSL-only suite to instrument key functions invoked by the pattern rewriter. While these micro-benchmarks are useful in their own right as performance measurements, they also facilitate further experiments. Their small size means they can be reasoned about at the bytecode level (chapter 4), providing useful information about the dynamism of the workload.

Chapter 4

Profiling at the bytecode level

Performance profilers are powerful tools that provide useful information about program control flow and hotspots, facilitating performance optimisation. Our research requires detailed information instrumenting Python's internal evaluation loop to draw comparisons with the C++ language runtime and characterise the cost of dynamism. However, existing profilers do not provide information at this granularity, motivating our work in this area. In this chapter we present ByteSight, a novel tool for performance profiling at the bytecode level, allowing us to look deeper into the performance characteristics of highly dynamic programs whose implementation cannot easily be deferred into lower level languages.

Recent developments to CPython's runtime motivate collecting more fine-grained profiling information. For example, the specialising adaptive interpreter rewrites bytecode at runtime into quickened forms, and the baseline JIT substitutes bytecode for tier two micro-operations – both yielding performance characteristics which cannot be reasoned about with function or line level instrumentation. In addition to this, bytecode performance profiling information helps provide a key missing data point when examining the impact of program dynamism. However, to the author's knowledge, no profilers exist which support measurement at this granularity (Related work, section 8.2). One reason for this conspicuous absence of bytecode level profilers is the difficulty of measuring their very short execution times, in the order of the highest resolution system counter. This difficulty is further worsened by the bytecode dispatch and evaluation being deeply interleaved within the interpreter's execution loop.

ByteSight, our novel tool for the performance profiling of CPython bytecode, addresses this gap in the existing provision. ByteSight is a tracing profiler which operates on the bytecode level of the Python interpreter. It is implemented natively in Python using only the standard library, and its source code is available under the MIT licence on GitHub.[1]It does not require patches to Python's language implementation. This makes it portable, robust across language versions, and simple to install from the Python Package Index

[1] `https://github.com/EdmundGoodman/bytesight`

(PyPI). In this chapter, we discuss its implementation and provide examples of its use.

4.1 Implementation

By virtue of the flexibility and dynamism of its interpreter's implementation, CPython provides a wide variety of opportunities for instrumenting and introspecting running code. One example of this is the standard library function `sys.settrace`, which associates dynamic, user-defined callback functions with the dispatch of key virtual machine events. These include function calls, line execution, handling exceptions, and even individual bytecode operations (interchangeably referred to as opcodes in the documentation). This callback function receives the event type along with the CPython frame and code objects currently being evaluated by the interpreter, facilitating precise instrumentation of the internal operation of the interpreter. Other possibilities for collecting this information include making custom patches to the CPython implementation, or leveraging the `LLTRACE` feature of the debug build of Python. However, the former is specific to individual language versions, and the latter has a verbose output containing no performance information. Furthermore, both requiring re-compiling the CPython implementation from source, precluding simple installation by users from Python Package Index (PyPI). As such, we chose to leverage `sys.settrace`, accepting the challenges associated with its measurement for benefits it provides in portability and robustness across language versions.

Our tool captures bytecode level profiling information through a custom callback function which records a sequence of timestamps associated with the traced events. From this set of timestamps, we can calculate the execution duration of each emitted opcode. This constitutes profiling information of a higher granularity than existing Python performance profilers. In spite of its name, we leverage the `sys.settrace` function as opposed to its sister function `sys.setprofile`. We do this because `sys.settrace` has supported emitting trace events for opcodes since Python 3.7, whereas `sys.setprofile` only emits events at the function granularity. This approach of leveraging CPython's API provides benefits of robustness across versions and easy installation, but also causes a number of challenges for accurate and precise measurement of very short events.

These challenges come as a result of the callback function itself being a callable Python code object. As such, the infrastructure to invoke and evaluate this function has a significant performance cost, in many cases taking longer than the opcode it is instrumenting. To mitigate this, our custom tracing function (Listing 1) records the timestamp at the earliest point 1 and the latest point 4, allowing the majority of its overhead to be excluded. Properties of the frame are then set to emit events for opcodes but not lines 2, further avoiding extraneous overhead. Finally, a tuple of timestamps for the end of the last trace function and the start of the current one are recorded 3, upper bounding the duration of opcode execution. This sequence of recorded timestamps can then be used in combination with domain knowledge of the language runtime to calculate individual opcode durations.

```python
def _trace__collect_event_timestamps(
    self, frame: types.FrameType, event: str, _arg: Any
) -> Callable[..., Any] | None:
    """Trace function to collect opcode timestamps."""
    now_timestamp = perf_counter() 1

    frame.f_trace_lines = False 2
    frame.f_trace_opcodes = True

    if event == "opcode":
        self._timestamps.append( 3
            (
                self._prev_event_timestamp,
                now_timestamp,
            )
        )
        self._prev_event_timestamp = perf_counter()  4

    return self._trace__collect_event_timestamps
```

Listing 1: Trace callback function generating a sequence of timestamps instrumenting opcode events.

4.1.1 CPython internals

In order to infer opcode durations from the event timestamps emitted tracing function, we must first examine the language runtime implementation with which it is co-designed. In this section, we present a view of CPython 3.10's implementation as a basis for our profiler. In more recent Python versions such as CPython 3.13, aspects of the tracing infrastructure have been refactored. However, since both the API provided by the standard library and the position of the tracing logic in the evaluation loop remain the same, the procedure remains applicable to these newer versions. Code objects in CPython are executed by the `_PyEval_EvalFrameDefault` function (Listing 4.1a). This begins with initialisation to ready for evaluation a, then enters an unbounded evaluation loop b. This loop decodes the next bytecode instruction c, then switches on it to select the appropriate logic to execute d, and repeats until an exception is thrown or the code object terminates. Having an understanding of the evaluation loop, we can next discuss how it is instrumented.

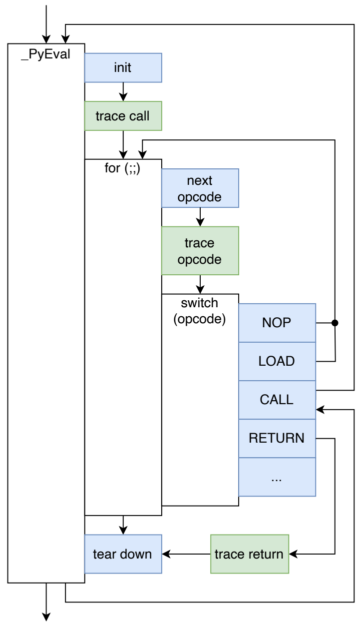
```c
PyObject* _PyEval_EvalFrameDefault(PyThreadState *tstate, PyFrameObject *f, int throwflag) {
    // ... declarations and initialization of local variables, macros definitions, call depth handling, ...  a
    // ... code for tracing call event

    for (;;) {  b
        // NEXTOPARG() macro  c
        _Py_CODEUNIT word = *next_instr;
        opcode = _Py_OPCODE(word);
        oparg = _Py_OPARG(word);
        next_instr++;

        // ... code for tracing opcode events

        switch (opcode) {  d
            case TARGET(NOP) {
                FAST_DISPATCH();
            }
            case TARGET(LOAD_FAST) {
                // ... code for loading local variable
            }
            // ... 117 more cases for every possible opcode
        }
    }
    // ... termination
}
```

(a) C implementation snippets, derived from an explanation of the bytecode evaluation by Victor Skvortsov [31].



(b) Control flow between evaluation (blue) and tracing (green).

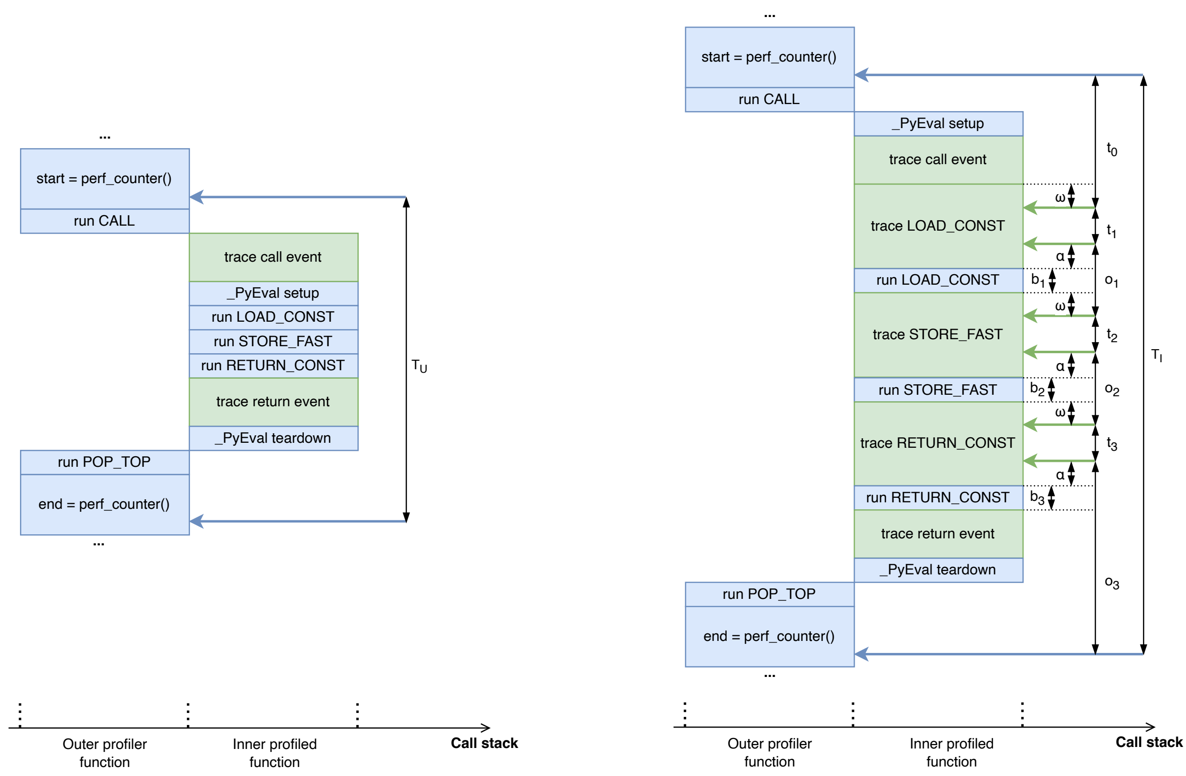
Listing 4.1: Overview of the evaluation loop of CPython 3.10, showing components relating to bytecode evaluation and tracing.

CPython's tracing mechanism works in two phases: registering the callback function, and the invocation of this callback function from the evaluation loop. To register the callback function, users can invoke `sys.settrace`, a standard library function binding to the C implementation of `sys_settrace` in `sysmodule.c:`. This in turn invokes `_PyEval_SetTrace` in `ceval.c`, which updates two fields on the Python Global Interpreter Lock (GIL) thread state struct: `c_traceobj` and `c_tracefunc`. The former is a callable Python code object for the callback function, and the latter points to a "trampoline" function, which invokes this code object with the current frame information. This trampoline is then invoked when events occur in the evaluation loop, including at the start of the frame evaluation for functions, after each opcode is extracted, and on returning from the function (green blocks in Figure 4.1b).

4.1.2 Inferring bytecode duration

Given both the sequence of timestamps and an understanding of their place in CPython's evaluation loop, we can infer our profiler's goal of the time taken to execute each opcode. One way to visualise this is by flattening the block diagram of the evaluation loop

(Figure 4.1b) and annotating it with timestamp measurements (Figure 4.2). From this, we can then construct a system of equations relating the measurements, and derive the durations of each opcode.



(a) Timing function runtime without opcode event tracing.

(b) Timing function runtime, including recording timestamps at the start and end of each opcode event.

Figure 4.2: Timing Python's evaluation loop with and without instrumentation of opcode events using `sys.settrace`.

By examination of where trace timestamps are recorded, we can see that the duration of the n$^{\text{th}}$ bytecode instruction, b_n, is equal to the difference between the recorded timestamp pairs, o_n, and the fixed overheads before and after the tracing function, α and ω respectively (Equation 4.1).

$$b_n = o_n - (\alpha + \omega) \tag{4.1}$$

Furthermore, the measured runtime of the instrumented function, T_I, is equal to the sum of the uninstrumented runtime T_U and the overhead incurred by tracing (Equation 4.2).

$$T_I = T_U + \sum_n (\alpha + \omega + t_n) \tag{4.2}$$

As such, we calculate the sum of the overheads before and after the tracing function, $\alpha + \omega$, under the assumption that they are fixed (Equation 4.3). This assumption is justified by the calling infrastructure for the tracing function being the same across all opcodes.

$$\alpha + \omega = \frac{T_I - T_U - \sum_n t_n}{n} \tag{4.3}$$

Finally, we can combine this with our initial observation to infer our goal of opcode duration (Equation 4.4).

$$b_n = o_n - \frac{T_I - T_U - \sum_n t_n}{n} \tag{4.4}$$

Beyond the careful co-design of the tracing measurement logic with CPython's implementation, there are a number of confounding effects which must be mitigated to ensure accurate measurement. Firstly, for the profiling information to be useful, the resolution of the most accurate system clock must be sufficient to resolve differences bytecode execution time. On our experimental hardware (subsection 3.1.1) this was true, having a 1ns timer able to resolve differences in opcodes taking around 10ns to execute. However, this is not the case for modern Apple Silicon devices. Their most accurate system timer, `mach_absolute_time` [32], has a resolution of only 40ns and is hence unable to resolve individual bytecode instruction durations of approximately 10ns. This is a physical limitation on measuring such necessarily fast events, and as such can only be resolved by selecting appropriate hardware. Secondly, the CPython language runtime periodically runs housekeeping tasks such as garbage collection, disrupting the flow of bytecode execution and hence adding random noise to our measurements. These can effects can be minimised using techniques from existing performance measurement work such as `timeit` [33] or `pyperf` [34], for example by disabling garbage collection for the duration of profiling. Finally, we repeat and aggregate our experimental measurements for statistical confidence, further minimising machine noise to ensure clean and reliable profiling results.

4.2 Example usage

Having implemented our profiler, we can demonstrate its capabilities on an example workload (Listing 4.3). Each traced event is displayed on its own line, in combination showing the exact sequence of instructions performed by the interpreter when evaluating the function. Function invocations, such as calling `inner_function` x, are indented by their call stack depth for easy readability. In addition to this, bytecode instructions are formatted following the convention of the standard library, but are annotated with their duration in a comment on the right-hand side of the trace y.

```python
import bytesight

def inner_function(
    x: int | str | float
) -> None:
    assert x

def example_function():
    inner_function(1)
    pass
    _x = perf_counter()

bytesight.profile_bytecode(
    example_function
)
```

(a) Python program.

```
// ======= example:8 `example_function` ========
// >>> inner_function(1)
7           0   LOAD_GLOBAL          0    (inner_function)    // 15    ns
            2   LOAD_CONST           1    (1)                 // 15    ns
            4   CALL_FUNCTION        1    ()                  // 31    ns

    // ======== example:3 `inner_function` =========  x
    // >>> assert x
    4           0   LOAD_FAST            0    (x)             // 13    ns
                2   POP_JUMP_IF_TRUE     4    (to 8)          // 13    ns
            >>  8   LOAD_CONST           0    (None)          // 12    ns
                10  RETURN_VALUE              ()              // 31    ns
    // =============================================

            6   POP_TOP                   ()                  // 16    ns
// >>> pass
8           8   NOP                       ()                  // 15    ns  y
// >>> _x = perf_counter()
9           10  LOAD_GLOBAL          1    (perf_counter)      // 15    ns
            12  CALL_FUNCTION        0    ()                  // 17    ns
            14  STORE_FAST           0    (_x)                // 14    ns
            16  LOAD_CONST           0    (None)              // 13    ns
            18  RETURN_VALUE              ()                  // 28    ns
// =============================================
```

(b) Profiler output.

Listing 4.3: Output of the bytecode profiling tool for a simple Python program, showing the sequence of dispatched bytecode and their individual execution times.

ByteSight provides straightforward mechanism to show the bytecode trace of any Python function. This facilitates debugging control flows in highly dynamic code, which often suffer from "spooky action at a distance", and is educational for the internal workings of Python's interpreter. This functionality is easily accessible by installation from PyPI, contrasting previous approaches which required either re-compiling the Python interpreter or manually implementing the same approach each time. Beyond this, the profiling information of the duration of each opcode is not provided by any existing tools to the author's knowledge. This information provides insight into the relationship between performance and dynamism in Python, and is more generally helpful for understanding and optimising performance critical code which cannot be deferred to a low-level language through Foreign Function Interface (FFI) bindings. In the context of our research, this tool unblocks our work in later chapters characterising and optimising the performance of xDSL.

Chapter 5

Manual specialisation of xDSL

In chapter 3, we constructed a set of experiments to empirically compare the current performance of the xDSL and MLIR compiler frameworks. Our overall aim is to use these experiments to contrast the performance of static and dynamic languages for the implementation of user-extensible compiler frameworks, using MLIR and xDSL as proxies for these two categories respectively. However, the experiments so far measure a combination of the effects of dynamism in the language runtime and implementation details of the framework, with the former obscuring our measurements of the latter. In order to use the frameworks for our aims, our measurements must be as independent of implementation details as possible.

In this chapter, we investigate the performance implications of language dynamism in user-extensible compiler frameworks by examining the impact of specialisation on the xDSL framework. By manually eliminating extraneous computation for specific workloads, we establish a performance baseline for pattern rewriting in xDSL that more accurately reflects the inherent costs of dynamism, independent of implementation overheads. To do this, we manually identify and remove computation performed by the framework which is unnecessary to its execution of this workload. For example, the framework may calculate and check values that are known as runtime invariants for the selected workload, for instance when checking properties of attributes known to be empty. Since this checking does not contribute to the workload's behaviour, it can be removed. Another example is functionality which improves the expressivity of the framework's API, such as runtime type checks to support polymorphism of arguments. Since the workload only uses a single facet of this API, this can also be removed. By eliminating the performance overheads from these unnecessary computations, we reach an implementation representative of the best-case performance of Python for this workload. This specialisation process yields an efficient pattern rewriting workload using xDSL's data structures. This demonstrates the best-case performance of CPython as a proxy for dynamic languages for such applications, facilitating comparison with C++ for insight into the relationship between performance and dynamism in user-extensible compiler infrastructures.

5.1 Micro-benchmarks

An important component of our experimental suite is micro-benchmarks, measuring the performance of procedures fundamental to xDSL and MLIR. Re-using these micro-benchmarks for specialisation allows the isolation of specific dynamic language features, and further provides a mechanism to precisely measure the impact of specialisation. By manually examining the traces of their execution, we can identify and eliminate any unnecessary computation in the current implementation. Having done this, we re-run the micro-benchmarks to quantify the performance of the specialised implementation, facilitating comparison of the language runtime only for compiler framework workloads.

5.1.1 Operation instantiation

The first micro-benchmark discussed in section 3.3 was instantiating operations, one of the central data structures in both xDSL and MLIR. The idiomatic way to do this in xDSL is directly instantiating a new Python object. In xDSL, this process includes a large amount of logic (Figure 5.1, top trace), including verifying attributes and building properties of the operation. However, in many instances this verification is unneeded as it is already known as a runtime invariant, and the building process can similarly be simplified as the structure can be inferred. As such, specialisation to remove this overhead can vastly reduce the complexity and hence improve performance (Figure 5.1, bottom trace).

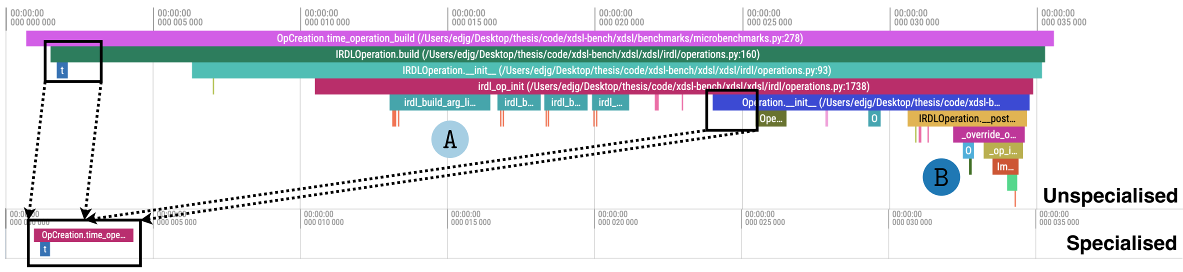


Figure 5.1: The unspecialised `EmptyOp` constructor (top) has a high overhead, including checking known invariants and building empty properties. This overhead can be avoided through specialisation to construct the same operation (bottom).

Examining the above trace, we can visually identify components which take a large proportion of runtime, and may represent unnecessary computation. For example, the xDSL constructor for `IRDLOperations` invokes logic with significant overhead building and filtering properties of the operation A. However, in this workload of the empty operation, this logic does not change the constructed object. Because of this, eliding this logic results in a specialised version of the operation which generates the same output using less computation by applying domain knowledge. In addition to this, MLIR's core operations are not IRDL based, unlike xDSL, meaning this overhead is implementation specific. Similarly, the post-constructor includes logic for context-managed builders B, which is again unrelated to the empty operation case and not present in MLIR. To avoid this overhead through specialisation, we move from the idiomatic constructor which initialises values to their defaults (Listing 5.2a) to directly modifying xDSL's underlying data structures (Listing 5.2b), where we initialise the defaults explicitly and without unneeded computation. Finally, we leverage an implementation detail of CPython's data model: the `__slots__` attribute [35]. Slots store attributes in a fixed-size array on the object with direct indexing instead using the default `__dict__` mapping from attribute name to value, optimising both lookup speed and space.

```
EmptyOp()
```

(a) Instantiation with constructors.

```
empty_op = EmptyOp.__new__(EmptyOp)
empty_op._operands = ()
empty_op.results = ()
empty_op.properties = {}
empty_op.attributes = {}
empty_op._successors = ()
empty_op.regions = ()
```

(b) Instantiation by direct manipulation of xDSL's data structures.

Listing 5.2: Unspecialised (left) and specialised (right) approaches to instantiating an `EmptyOp` differ in both length and performance.

Through specialising the implementation to the workload of this micro-benchmark we can achieve significant performance improvements of up to 26× (Table 5.1). A proportion of this speedup is captured by optimisations which C++ is able to make ahead-of-time, but that are invalidated by Python's dynamism. The most salient of these possible optimisations is function inlining calls for the constructor implementation, which would eliminate a significant overhead visible in the Python trace (Figure 5.1). In addition to this, there are further opportunities for optimisations such as instruction combination for more efficient computation, and dead code elimination for compile time invariants [36]. However, the performance improvement comes at the cost of a significantly more complex implementation, contrary to xDSL's design goals of a simple expressive API. We can quantify one cause of this improvement as the specialised code executing fewer, faster bytecode instructions by leveraging our novel bytecode profiling tool (full traces in Appendices E.1, E.2). For example, specialisation reduces the number of bytecode instructions executed from 466 to only 29. Function calls have a much longer duration than other instructions, with their mutation of the call stack taking up to three times longer than loading from a variable. Again leveraging our tool, we can see that specialisation reduces the number of function calls from 65 to 5, further contributing to the performance uplift. By examination of its emitted bytecode (Listing E.2), we can be confident the specialised implementation is near-optimal within the constraints of Python's runtime, as it only constructs the object and sets required fields, performing no other extraneous logic.

Table 5.1: Specialising `EmptyOp` instantiation in xDSL yields performance uplifts of up to 34×, only 2.5× slower than the MLIR reference implementation.

MLIR [ns]	xDSL [ns]	Optimised xDSL [ns]
153 ± 0.5	12700 ± 1810	378 ± 42

5.1.2 Operation trait checks

The second micro-benchmark discussed involved checking traits on operations, and was measured to perform 80× worse in xDSL than MLIR. Similarly to the operation instantiation micro-benchmark, this slow-down comes as a result of a high proportion of method's logic not being required by the core functionality (Figure 5.3, top trace). In this case, this logic improves the expressivity of the API, supporting both concrete objects and types as arguments, along with handling the edge case of unregistered operations. However, this logic lies on the hot path of execution, and again these properties are known as runtime invariants. As such, specialisation can again be leveraged to remove this implementation overhead (Figure 5.3, bottom trace).

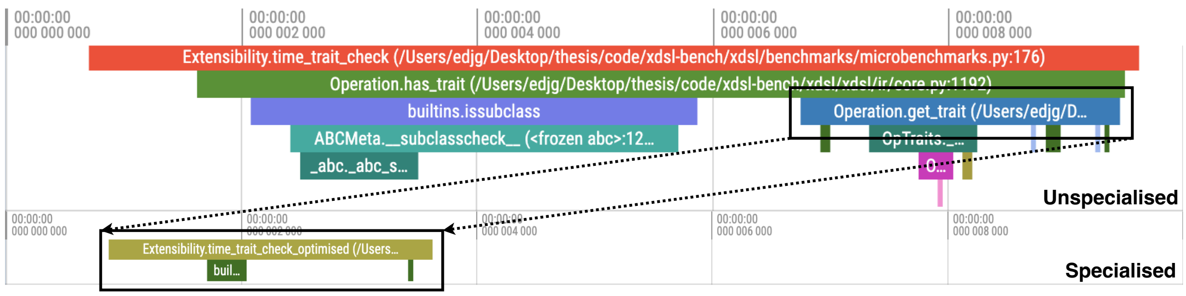


Figure 5.3: `issubclass` and `isinstance` checks, type `casting`, and constructing iterators constitutes over three quarters of xDSL's unspecialised `has_traits` runtime (top). Specialisation to narrow the interface and optimisation to avoid extraneous work on hot paths can significantly accelerate `has_trait` (bottom).

Examining the implementation of this micro-benchmark (Listing 5.4), we can again identify opportunities for specialisation and optimisation. The execution trace shows that checking whether an operation is unregistered 1 accounts for over one third of the micro-benchmark runtime. Through specialisation under the runtime invariant that all operations are registered, this check can be removed – reducing computation. Furthermore, this inspires a general optimisation to xDSL to avoid this check by overloading `UnregisteredOp.has_trait` rather than checking on all code paths. In addition to this, function invocation in Python incurs a performance cost due to the overhead of modifying the stack frame. As such, inlining the call to `get_trait` 2 is another beneficial specialisation. Finally, argument types are dynamically checked 3 and cast 4, incurring a performance overhead. However, the type of these arguments are runtime invariants of the caller, so the checks can be specialised away.

```python
@classmethod
def has_trait(
    cls,
    trait: type[OpTrait] | OpTrait,
    *,
    value_if_unregistered: bool = True,
) -> bool:
    from xdsl.dialects.builtin import UnregisteredOp  1
    if issubclass(cls, UnregisteredOp):
        return value_if_unregistered

    return cls.get_trait(trait) is not None  2
```

(a) Outer `has_trait` method.

```python
@classmethod
def get_trait(
    cls,
    trait: type[OpTraitInvT] | OpTraitInvT
) -> OpTraitInvT | None:
    if isinstance(trait, type):  3
        for t in cls.traits:
            if isinstance(t, cast(  4
                type[OpTraitInvT], trait
            )):
                return t
    else:
        for t in cls.traits:
            if t == trait:
                return cast(OpTraitInvT, t)
    return None
```

(b) Inner `get_trait` method.

Listing 5.4: xDSL methods implementing trait check functionality.

Having specialised xDSL's implementation, we can draw direct comparison between its implementation (Listing 5.5a) and MLIR's (Listing 5.5b), to better understand their relative performance characteristics. In contrast to the original, the specialised implementation directly matches MLIR, with the only difference being approach to checking RTTI. As such, it is well-suited for comparing the performance characteristics of static and dynamic languages independent of implementation details.

```python
for t in OP.traits._traits:
    if isinstance(t, TRAIT):
        return True
return False
```

(a) xDSL's modified `has_trait` method.

```cpp
TypeID traitIDs[] = {TypeID::get<Traits>()...};
for (unsigned i = 0, e = sizeof...(Traits); i != e; ++i)
    if (traitIDs[i] == traitID)
        return true;
return false;
```

(b) MLIR's `has_trait` method.

Listing 5.5: xDSL and MLIR methods searching trait arrays.

Through specialisation, we achieve a 8× speedup over the original implementation (Table 5.2). However, this again incurs a cost to xDSL's expressivity, sacrificing polymorphic support for checking traits of both object types and instances. As with operation instantiation, this is contrary to xDSL's design goals. Furthermore, this speedup is less dramatic than operation instantiation, as a result of having less overhead in the original implementation, but can be reasoned about in the same manner with our novel profiling tool (full traces in Appendices E.3, E.4). As before, specialisation reduces the number of bytecode instructions from 89 to 35, with the number of high-overhead `CALL` instructions dropping from 11 to 2.

Table 5.2: Specialising trait checking in xDSL yields performance uplifts of up to 8×, 20× slower than the MLIR reference implementation.

MLIR [ns]	**xDSL** [ns]	**Optimised xDSL** [ns]
11.7 ± 0.5	1920 ± 695	239 ± 35

5.2 Pattern rewriting

Having demonstrated specialisation as an approach to examine the performance bound of a language runtime for micro-benchmarks, we can further apply it to real-world workloads such as constant folding. In this section, we specialise a simple pattern rewriter: constant folding over integer addition. These specialised implementations can then be taken as performance baselines for their respective languages, and further analysed through bytecode tracing or disassembly to understand their implementation.

5.2.1 Constant folding

For our first benchmarks of the two frameworks, we measured the end-to-end performance of canonicalization passes applied to an IR with many foldable constants (section 3.1.2). However, canonicalization passes are fairly complex, encompassing a suite of common optimisations, many of which are not applicable to our IR workload. This complexity is detrimental to manual specialisation, resulting in more code requiring transformation by hand. As such, we implement a simple pattern rewriting pass for constant folding over the addition of integers (Listing 5.6). This pass captures common idioms in pattern rewriting, making it a fair proxy for more complex workloads, but is also sufficiently simple to rewrite by hand into a fully specialised form. The rewriter first matches against the integer addition operation to fold 1, exercising operation type checks. Next, it checks the invariant of our workload that both addition operands are constant 2, exercising trait lookups. After this, it gets the value of each operand, exercising operation properties 3. Finally, it replaces the matched operation with a new one, exercising the rewriting mechanism 4.

```python
def match_and_rewrite(self, op: Operation,
↪ rewriter: PatternRewriter, /):
    # Only rewrite integer add operations 1
    if not isinstance(op, AddiOp):
        return

    # Ensure both operands are constants 2
    lhs_op = op.operands[0].op
    rhs_op = op.operands[1].op
    if lhs_op.has_trait(ConstantLike) or
    ↪ rhs_op.has_trait(ConstantLike):
        return

    # Calculate the result of the addition 3
    lhs = lhs_op.value.value.data
    rhs = rhs_op.value.value.data
    folded_op = ConstantOp(
        IntegerAttr(lhs + rhs, op.result.type)
    )

    # Rewrite with the calculated result 4
    rewriter.replace_matched_op(
        folded_op, [folded_op.results[0]]
    )
```

(a) xDSL implementation.

```cpp
// Only rewrite integer add operations 1
LogicalResult matchAndRewrite(arith::AddIOp op,
↪ PatternRewriter &rewriter) const override {

  // Ensure both operands are constants 2
  arith::ConstantOp lhsConstOp =
  ↪ op.getLhs().getDefiningOp<arith::ConstantOp>();
  arith::ConstantOp rhsConstOp =
  ↪ op.getRhs().getDefiningOp<arith::ConstantOp>();
  if (!lhsConstOp || !rhsConstOp) {
      return failure();
  }

  // Calculate the result of the addition 3
  auto lhsAttr =
  ↪ lhsConstOp.getValue().dyn_cast<IntegerAttr>();
  auto rhsAttr =
  ↪ rhsConstOp.getValue().dyn_cast<IntegerAttr>();
  if (!lhsAttr || !rhsAttr) {
      return failure();
  }
  APInt lhsValue = lhsAttr.getValue();
  APInt rhsValue = rhsAttr.getValue();
  APInt result = lhsAttr.getValue() +
  ↪ rhsAttr.getValue();

  // Rewrite with the calculated result 4
  auto resultType = op.getType();
  auto foldedValue =
  ↪ rewriter.getIntegerAttr(resultType, result);
  rewriter.replaceOpWithNewOp<arith::ConstantOp>(op,
  ↪ resultType, foldedValue);
  return success();
}
```

(b) MLIR implementation.

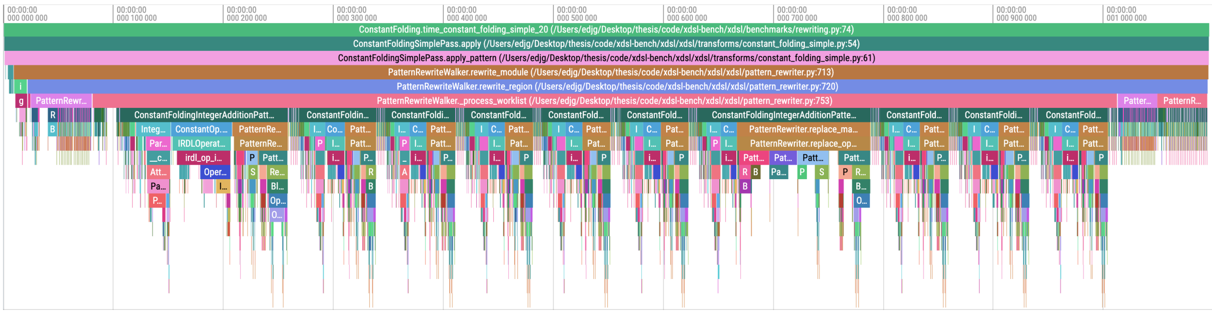
Listing 5.6: Implementations of pattern rewriters for constant folding over integer addition.

In order to draw fair comparisons between the two frameworks, their implementations of the constant folding pattern rewrite must be equivalent. As such, we provide implementations for both xDSL (Listing 5.6a) and MLIR (Listing 5.6a). Measuring the MLIR implementation is complicated by the fact that its pattern rewriting infrastructure implements constant folding by default. To mitigate this, we manually excise this functionality from the MLIR framework to ensure that our pattern rewriting kernel is the implementation being measured. Having constructed these implementations, we can follow the same specialisation process introduced for the micro-benchmarks to bring the xDSL's performance closer to the best-case for that workload in the Python language.

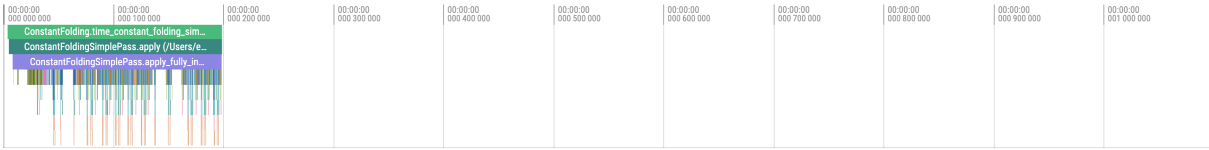
5.2.2 Specialisation

Although our constant folding pass is less intricate than existing passes such as canonicalization, it is still much more complex than any of the above micro-benchmarks (Figure 5.7a).

Instead of one or two helper functions in xDSL's API, aspects of the pattern rewriter such as replacing the matched operation have a deep call stack, invoking methods to set many aspects of the IR's object representation. As such, the first step in the specialisation process is manually inlining this call stack, which provides two benefits. Firstly, it avoids the non-negligible overhead of function calls discussed in previous micro-benchmarks, and further worsened by the deep call stack of the more complex implementation. Secondly, it reveals logic that is redundant for the implementation of this workload, which is otherwise hidden across the boundaries of xDSL's API. The inlined implementation can then be specialised to remove this redundant logic, and further leverage runtime invariants of the integer constant addition workload where possible. This yields a specialised implementation with reduced overhead (Figure 5.7b).



(a) xDSL's existing pattern rewriting infrastructure has a high performance overhead, executing many extraneous operations with a deep call stack.



(b) Through specialisation, this overhead can be significantly reduced, performing the exact workload and only invoking inbuilt functions such as set operations and `isinstance` checks.

Figure 5.7: Profiling constant folding pattern rewrites of integer addition before (top) and after (bottom) specialisation with `viztracer` reveals unnecessary computation.

5.2.3 Performance improvement

Through specialising the constant folding implemenation, we achieve a 8× speedup over the original implementation (Figure 5.8). However, this performance uplift comes at the cost of ease of implementation, increasing the constant folding implementation length by 7×. As with the specialised micro-benchmarks, we can leverage our novel bytecode profiling tool to explain this improvement in terms of dispatched bytecode instructions, with the workload dropping from 5465 to 977 instructions required to implement its functionality. By examining the implementation and its emitted bytecode instructions, we are confident that it is close to optimal for the workload within the context of xDSL's data structures. In combination with the modifications made to MLIR to similarly avoid extraneous computation, these two implementations are both directly comparable and complex enough to be representative of real-world workloads. This facilitates their later use for exploring the impact of dynamism on such workloads. Comparison cannot be made with the unmodified version, as MLIR's implementation of pattern rewriting applies constant folding within the framework machinery, clobbering any user-defined constant folding pass.

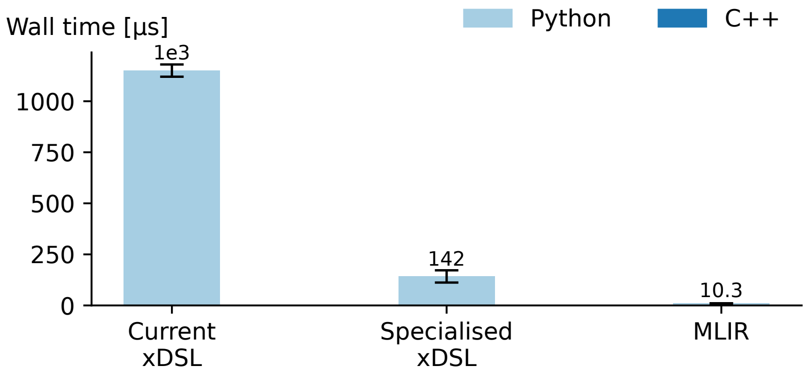


Figure 5.8: Specialisation of constant folding over integer addition in xDSL yields performance uplifts of up to 8×, 14× slower than the MLIR reference implementation.

5.2.4 Optimisations

The majority of changes made during specialisation are applicable only to their specific workload. However, some are more generic to any use case of xDSL, and as such can be applied as stand-alone optimisations. Unfortunately, xDSL is a large codebase, over 150,000 lines of Python, and as such it is out of scope of this thesis to improve the performance of all of it. Despite this, we demonstrate that the specialisation process lead to generalisable optimisations with measurable performance impact. Through this, we enable performance improvement over time, which can be quantified using our previously introduced measurement infrastructure (subsection 3.1.3). In this section, we discuss two salient examples of these generalised optimisations.

During the specialisation process of trait checking (subsection 5.1.2), we saw that the implementation of xDSL's `has_trait` helper function introduces significant overhead by checking the infrequent case of unregistered operations on the hot path of execution. Since this functionality was not used by the target workload, this could be elided by specialisation for performance. However, it also reveals an optimisation which can be applied to the general case, resulting in a 2.1× speedup on micro-benchmarks and 1.15× speedup within the specialised constant folding workload[1]. Similarly, the specialisation of operation instantiation (subsection 5.1.1) revealed the performance impact of managing the scope of implicit builders, even when they are not being used. As before, this can be generalised to change xDSL's approach from always invoking logic to handle implicit

[1] `https://github.com/xdslproject/xdsl/pull/4242`

builders irrespective of whether they are used, to only for operation instantiations which use them, resulting in a 1.06× speedup on micro-benchmarks and 1.01× speedup within the specialised constant folding workload[2].

5.3 Summary

To evaluate whether dynamic languages approach the performance of ahead-of-time compiled languages for implementing user-extensible compiler frameworks, we empirically measure Python's best-case performance for pattern rewriting tasks. By examining specialised implementations of micro-benchmarks and real-world pattern rewriting workloads, we demonstrate that xDSL can achieve performance within 14× of equivalent MLIR implementations. This represents an 8× improvement over xDSL's original implementation, capturing optimisations which can be made automatically by ahead-of-time compilers but not dynamic language runtimes. This specialisation process both eliminates workload-specific redundant computation and avoids costs associated with function invocation, while further revealing generalisable optimisations that benefit xDSL beyond individual workloads. Furthermore, this best-case performance analysis establishes an empirical upper bound for dynamic language performance in compiler infrastructure. In the next chapters, we leverage this performance ceiling as a baseline to examine the effect of both recent improvements to CPython's runtime, and the cost of dynamism in comparison with statically compiled languages.

[2] `https://github.com/xdslproject/xdsl/pull/4163`

Chapter 6

CPython optimisations for xDSL

Python Enhancement Proposal (PEP) 659 asserts that "Python is widely acknowledged as slow" [37]. This comes partially as an inherent trade-off from the benefits of its interpreted runtime and expressive dynamic semantics, meaning it cannot achieve the general-purpose performance of ahead-of-time compiled languages such as C++ or FORTRAN. However, it is feasible for Python implementations to be competitive with fast implementations of other scripting languages with similar trade-offs, such as JavaScript's V8 or Lua's LuaJIT. The Faster CPython project is an attempt to achieve this goal in Python's reference implementation. Over the course of the recent CPython major versions, new optimisations have been gradually added as part of this project, resulting in incremental performance gains (Table 6.1). This section discusses the details of these optimisations, and their effect on user-extensible compiler workloads in xDSL.

6.1 Specialising adaptive interpreter

Simple interpreters of dynamic languages use generic instructions which can express functionality over a wide array of types. However, this incurs an overhead selecting the specific implementation for the current type at runtime. In 2009, Williams et al. introduced the concept of instruction specialisation in their paper "Dynamic Interpretation for Dynamic Scripting Languages" [14]. This proposes a novel dynamic IR for Lua, whose operations can be specialised to a more performant implementation based on types and control flow encountered at runtime, claiming an average speedup of 1.3×. Following this, Mark Shannon applied the idea to the Python language in his doctoral thesis "The construction of high-performance virtual machines for dynamic languages" [38], demonstrating its viability through the research Virtual Machine (VM) HotPy. This idea was realised in the CPython through Python Enhancement Proposal (PEP) 659's specialising adaptive interpreter, enabled by default in the 2022 minor version 3.11 release.

Table 6.1: Incremental improvements of the geometric mean of speedups across the PyPerformance benchmark suite (full results Appendix A), achieved by optimisations to the CPython interpreter since CPython 3.10.

Python executable		PyPerformance speedup	
Implementation	**Feature flag**	**Baseline**	**Previous**
CPython 3.10.17	None	1×	1×
CPython 3.11.12	None	1.241×	1.241×
CPython 3.13.3	None	1.312×	1.078×
CPython 3.13.3	Experimental JIT	1.295×	0.988×

The specialising adaptive interpreter implements this by replacing instructions which can be optimised by specialisation, such as the generic `LOAD_ATTR`, with an adaptive form, such as `LOAD_ATTR_ADAPTIVE`. Each adaptive instruction maintains an internal counter, incrementing it when the observed usage matches a specialisation opportunity, and decrementing when it does not. When this counter exceeds a threshold, the adaptive instruction replaces itself with the appropriate specialised version, in a process known as a quickening. For example `LOAD_ATTR_ADAPTIVE` becomes `LOAD_ATTR_INSTANCE_VALUE` when the attribute is an instance of the class for previous execution paths of the instruction. If the assumptions required for specialisation are violated later, then the interpreter can fall back to the original instruction implementation to ensure correctness. The optimisation is transparent to users, as no code changes are required, and the interpreter ensures correctness through its fall-back mechanism. The specialising adaptive interpreter's documentation claims a speedup of 10% – 60% [37], matching our recorded geometric mean improvement of 24% for the PyPerformance benchmarks (Table 6.1). This is substantiated by an ablation of the feature across the both previously discussed real-world constant folding workload and micro-benchmarks.

Table 6.2: Execution time per operation improves by a geometric mean speedup of 1.36× across micro-benchmarks and real-world workloads in xDSL when the specialising adaptive interpreter is used (CPython 3.11).

	Execution time [s]		
Workload	**CPython 3.10**	**CPython 3.11**	**Speedup**
Operation instantiation	3770 ± 916	2840 ± 883	1.33×
Specialised operation instantiation	477 ± 385	265 ± 372	1.8×
Trait checking	1920 ± 695	1710 ± 753	1.12×
Specialised trait checking	266 ± 34	191 ± 334	1.39×
Operation traversal	525 ± 3	432 ± 4	1.21×
Constant folding	57000 ± 1895	43000 ± 1340	1.33×
Specialised constant folding	6750 ± 1340	4620 ± 1075	1.46×

We measure the specialising adaptive interpreter to yield a non-negligible speedup of 1.38× across both micro-benchmarks and real-world workloads (Table 6.2). This speedup is greater than the 1.26× recorded across the PyPerformance suite. Since the specialising adaptive interpreter works by quickening highly dynamic instructions into specialised versions, this demonstrates that xDSL as a proxy for user-extensible compiler framework workloads is more dynamic than the baseline of PyPerformance workloads. From this proxy measurement, we hypothesise that user-extensible compiler framework workload is a highly dynamic one, motivating our later quantification of this (chapter 7). The documentation of PyPerformance states “the focus is on real-world benchmarks, rather than synthetic benchmarks, using whole applications when possible.” [20], further widening this conclusion to user-extensible compiler framework workloads being more dynamic than the average of a set workload representative of the real-world use of Python. In addition to this, the specialised versions of each workload yield a greater speedup after applying the manual specialisation process described in chapter 5. This shows a further benefit of specialisation in revealing optimisation opportunities by the language runtime.

6.2 Experimental JIT compiler

Another bottleneck of simple interpreted language runtimes is the overhead associated with dispatching and executing bytecode operations. To address this, just-in-time (JIT) compilation can translate frequently executed portions of bytecode into native machine code. This avoids interpreter overhead and allows for further low-level optimizations. CPython’s experimental JIT compiler is an implementation of the copy-and-patch machine code generation approach (introduced in subsection 2.6.1), which was added to CPython in 3.13 in 2024. This idea was finally realised in the CPython reference implementation through PEP 744’s experimental JIT, optionally enabled by a feature flag in the 2024 minor version 3.13 release.

To implement this, CPython introduced a new internal intermediate representation: tier two opcodes. Tier two opcodes are even more fine-grained that traditional bytecode and as such more amenable to copy-and-patch compilation. When compiling CPython, the LLVM toolchain is used to create small re-usable snippets of machine code for the target machine called stencils, which implement the functionality of these tier two opcodes. At runtime, when bytecode is executed frequently it is marked as hot. Hot bytecode is then translated into a sequence of these more granular tier two opcodes, and optimised by applying transformation passes. The JIT then translates each tier two opcode into executable machine code by copying and patching runtime information into the holes in the pre-compiled stencils. This machine code can then run directly on the processor, replacing the slower interpreter loop to execute the equivalent bytecode instructions. The JIT is experimental and disabled by default in CPython 3.13, but lays the groundwork for significant future performance enhancements. As such, its performance change for many workloads is minimal or even regressive. As with the specialising adaptive interpreter, we substantiate these claims by an ablation of the feature across the previously discussed real-world workload and micro-benchmarks.

Table 6.3: Execution time per operation improves by a geometric mean slowdown of 0.971× across micro-benchmarks and real-world workloads for CPython 3.13.3 when the experimental JIT is enabled.

	Execution time [s]		
Workload	**JIT disabled**	**JIT enabled**	**Speedup**
Operation instantiation	2990 ± 808	2980 ± 759	1×
Specialised operation instantiation	365 ± 32	391 ± 36	0.93×
Trait checking	1190 ± 553	1220 ± 573	0.98×
Specialised trait checking	221 ± 31	224 ± 30	0.99×
Operation traversal	394 ± 2	427 ± 2	0.92×
Constant folding	44600 ± 1330	47000 ± 1450	0.95×
Specialised constant folding	5400 ± 1225	5250 ± 1275	1.03×

In contrast to the specialising adaptive interpreter, the experimental JIT regresses performance for nearly all workloads (Table 6.3). This aligns with the results recorded for the PyPerformance benchmark suite, which similarly regress when the JIT is enabled. As discussed above, this comes as a result of the JIT's experimental nature, aiming to provide infrastructure for future optimisations rather than immediate gains.

6.3 Summary

In recent years, Faster CPython has introduced two key optimisations which leverage the runtime information as a result of dynamism in the Python interpreter. The first is the specialising adaptive interpreter, which profiles how generic bytecode instructions are used, allowing speculative quickening into a more specialised form. The second is the experimental JIT, which identifies hot code paths at runtime and translates them into optimised native machine code. Beyond these dynamic optimisations which we examine in detail, Faster CPython also makes other optimisations unrelated to dynamism in the language runtime, such as the tail-calling interpreter [39] and memory layout optimisations which further improve performance.

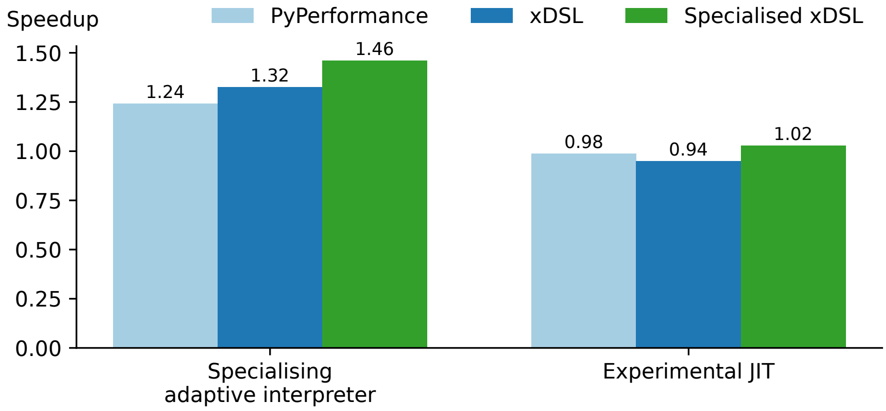


Figure 6.1: Highly dynamic workloads such as constant folding in xDSL exacerbate the performance impact of CPython optimisations leveraging runtime information, with manual specialisation further revealing optimisation opportunities.

Through our empirical measurement, we show that the pointer-chasing, variable control flow workload of the xDSL user-extensible compile framework exacerbates the impact of both of these optimisations leveraging runtime information (Figure 6.1). This can be seen by constant folding in xDSL having a greater speedup than PyPerformance with the specialising adaptive interpreter, but a worse slow-down with the experimental JIT. In addition to this, we show that specialisation of the workloads (chapter 5) reveals further optimisation opportunities, increasing the impact of both of these optimisations.

Chapter 7

Dynamism in compiler frameworks

A key difference between the Python and C++ runtimes is their degree of dynamism. The C++ runtime incurs overhead when dynamically dispatching functions (Figure 1.1, 3), which is worsened by prohibiting ahead-of-time performance optimisations. In contrast, Python dynamically evaluates each bytecode operation individually in its interpreter loop, incurring an overhead each time. As such, we expect the difference in performance between language runtimes (Figure 1.1, 4) to be smaller for more dynamic workloads. In this chapter, we quantify this difference by examining synthetic examples within static and dynamic language runtimes. We then apply this information to understand the difference in performance between pattern rewriting workloads using xDSL and MLIR through the lens of overhead incurred by dynamism.

7.1 Cost of dynamic dispatch

In static languages such as C++, the function calls can often be resolved at compile time. In contrast, dynamic languages such as Python must resolve the address of each function call at runtime, incurring an overhead. However, the address of some function calls can only be known at runtime, for example as a result of object polymorphism. This address then must be resolved during execution by the language runtime in both static and dynamic languages. Furthermore, this information being known only at runtime presents an optimisation boundary, precluding common rewrites such as function inlining which contribute to the performance of ahead-of-time compiled languages. We argue further that the difference between static and dynamic languages is reduced for workloads with insufficient information to resolve function addresses ahead of time. We justify this by examining the mechanisms of dynamic dispatch in Python and C++, and contrasting them through both synthetic examples and our micro-benchmark suite.

C++ uses a Virtual Method Table (vtable) mechanism for method polymorphism, a lookup table which is accessed at runtime through pointer indirection to retrieve the address for the virtual function implementation. In contrast, Python stores methods and attributes in the `__dict__` object attribute which is searched at runtime, checking parent classes if needed. While both use indirection for dynamic dispatch, Python's approach enables more dynamic behaviour like runtime meta-programming but comes with higher performance costs compared to C++'s more efficient vtable system. We can quantify the performance overhead of this vtable mechanism through a synthetic example (Listing 7.1).

```
class Base {
public:
    int func(int a, int b) { b
        return a - b;
    }
    __attribute__((noinline))
    int uninlinedFunc(int a, int b) { e
        return a - b;
    }
    virtual int virtualFunc(int a, int b) { a
        return a - b;
    }
};

class Derived : public Base {
public:
    int virtualFunc(int a, int b) override {
        return b - a;
    }
};
```

(a) Method definitions.

```
#include <stdlib.h>

int main(int argc, char *argv[]) {
    // Values known only at runtime c
    int a = atoi(argv[1]), b = atoi(argv[2]), c
    ↪  = atoi(argv[3]);

    // Setup
    int result = 0;
    Base baseObj;
    Derived derivedObj;
    Base* polyObj = c > 0 ? &baseObj :
    ↪  &derivedObj; d

    // Function invocations
    result += baseObj.func(a, b);
    result += baseObj.uninlinedFunc(a, b);
    result += polyObj->virtualFunc(a, b);

    return result;
}
```

(b) Method invocations.

Listing 7.1: Synthetic example of direct and dynamic method dispatch in C++.

This synthetic example exercises polymorphic methods which must be resolved dynamically using a vtable at runtime a, along with methods which can be statically resolved ahead of time during compilation b. A challenge when constructing this workload is providing data whose value is known only at runtime. We implement this by taking arguments from the command line c, as opposed to defining static variables which the compiler could reason about to inform ahead of time optimisations. This is necessary to exercise dynamic dispatch of functions, which requires a polymorphic object whose type is known only at runtime d. Since this simple synthetic example is amenable to compiler optimisations such as function inlining, we use variable attributes to hint to the compiler that certain methods should not be inlined e.

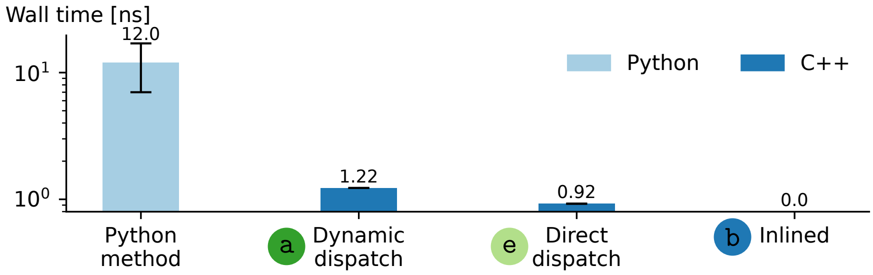


Figure 7.2: Dynamic dispatch generated by `clang -O3` incurs a 30% overhead in comparison with direct dispatch, but remains an order of magnitude more performant than CPython 3.10 method invocation.

Having constructed this synthetic example, we calculate the cost of each method invocation by measuring the runtime of each function and subtracting the runtime of its inlined implementation (Figure 7.2). Dynamic dispatch is 30% slower than direct dispatch, as a result of the overhead constructing and dereferencing through the vtable. Examining the disassembly (Appendix F), this overhead increases the instruction count from 4 to 14, for a total of 10 extra cycles on ARM Reduced Instruction Set Computer (RISC) machines. This observation matches the work of Driesen and Hölzle, who assert the "direct cost" of virtual function calls is up to 10.2 cycles for highly dynamic workloads [40, Figure 18.]. In addition to this Driesen and Hölzle acknowledge there is a further "indirect cost" associated with hidden optimisations, but do not characterise it. An example of such an optimisation hidden by runtime information is the inlining of the function implementation. This code motion represents the remaining 70% of the overhead of dynamic dispatch – significantly greater than the polymorphism machinery. Furthermore, this is a lower bound of this impact, as further optimisations such as vectorisation or dead code elimination could be revealed. Despite both these direct and indirect costs, Python function invocation remains an order of magnitude slower.

Functions which can only be dispatched at runtime are one way a workload can exhibit dynamism. An example of this in the domain of compiler frameworks is the `Operation` objects composing the IR being processed are necessarily only known at runtime, their methods such as verification and printing must be dispatched dynamically. In other cases, MLIR is optimised to avoid this cost. For example, the `TypeID::get<Traits>` function uses template meta-programming to monomorphise the generic function calls. However, this approach is only applicable when there is sufficient information at runtime.

7.2 Run-time type information

In a "A history of C++: 1979–1991", Stroustop states that the original C++ design "deliberately didn't include [mechanisms] for run-time type identification [as] they were almost always misused." [41]. Support for this functionality was later added in C++98 [42], including support for `dynamic_casts` checked a runtime, and getting the `typeid` of a polymorphic object. However, this incurs a runtime cost [43], and is brittle in the objects to which it can be applied. A such, LLVM reimplements a subset of this functionality, providing the `dyn_cast` method and `TypeID` data structure, aiming to "strike a balance between performance and the setup required to enable its use" [44]. This is another example of dynamic behaviour, which we again argue incurs additional runtime overhead and precludes optimisations in static languages, closing the gap with dynamic ones. As before, we justify this be examining the details of this mechanism, using our micro-benchmark suite.

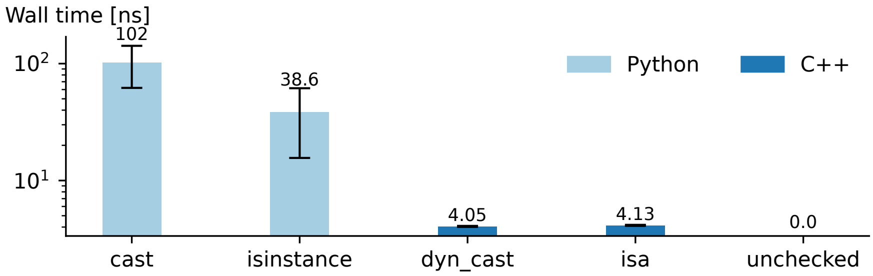


Figure 7.3: Both checking and casting LLVM RTTI have a similar runtime cost, contrasting Python, which is 10× slower for checking and 25× slower for casting.

We measure the duration of micro-benchmarks exercising xDSL and MLIR's RTTI operations (Figure 7.3). LLVM's RTTI implementation is much more performant than Python, leveraging C++ templates to defer as much computation as possible to compile time. In addition to this, type checking and casting have approximately the same performance cost, as they leverage similar mechanisms. In contrast, Python's `isinstance` function is faster than `cast`, as the former is implemented in C as a builtin function, whereas the latter is in Python in the standard library, hence relying on the dynamic dispatch machinery discussed above. However, the implementation of the `cast` function is the identity, doing nothing unless explicitly overridden. Python's dynamic duck typing [45] does not require restructuring data when casting, allowing casting operations and some type checks to be elided. Despite this, the `cast` function incurs a high overhead. In CPython, even empty function calls require at least two expensive bytecode operations, and they cannot be optimised away as they may be overridden dynamically at runtime.

7.3 Dynamic languages for user-extensible compilers

From the previous section we can see that dynamism incurs overhead and precludes ahead of time optimisations. This narrows the performance gap between dynamic and static languages for the implementation of this infrastructure, because the cost traditionally associated with Python's dynamic interpreted runtime is lessened by the commensurate dynamism of the user-extensible compiler framework workload. Contrasting the $60,000\times$ slowdown claimed of C++ implementations of structured workloads such as GEMM [15], specialised implementations of real-world xDSL workloads demonstrate Python can close this gap to only 10×. In combination with developer productivity being critical for modern compiler engineers to achieve their performance goals (chapter 2), this narrowed gap challenges the status quo of MLIR being implemented static, ahead-of-time compiled C++. Instead, xDSL's approach of using Python improves developer productivity with its expressive syntax and fast build times, empowering fast prototypes during development. We believe that this motivates the use of dynamic languages for the implementation of user-extensible compiler infrastructures.

Chapter 8

Related work

In this chapter, we survey existing literature analysing the performance of compiler frameworks and the Python language, along with the role of dynamism in programming language design. From this, we show the research gaps which this thesis aims to address.

8.1 Compiler framework performance

While compiler designers naturally focus on optimizing the performance of compiled code, the execution time of the compiler itself also has a significant impact on developer productivity. For very large projects such as Firefox, which contains over twenty million lines of code [46], small changes to compiler performance can result in minutes gained or lost for each compilation – significantly impacting developer productivity. Although there has been considerable engineering effort devoted to compiler performance, academic research continues to focus primarily on improving the compiled output rather than the compilation process itself. As such, researchers have not yet fully explored the field, with only a few academic studies examining the performance of compiler frameworks, the most salient of which we discuss below.

Lattner and Adve's original paper proposing LLVM contains a short evaluation of the framework's performance [2, Section 4.1.4]. In this evaluation, the authors compare the runtime of individual transformation passes against `gcc` optimisation level `-O3` across a variety of workloads. The results of this experiment [2, Table 2] report each of LLVM's transformation passes is at least two orders of magnitude faster than `gcc`'s end-to-end compilation across the tested workloads. This demonstrates that analysis and transformations can be performed efficiently, but has a critical flaw. By using end-to-end compilation time for `gcc`, the measurement includes time taken by non-transformation phases such as parsing, printing, and code generation, making it incomparable with the measurements of the transformation passes only for LLVM. Furthermore, since the paper's publication twenty years ago, LLVM has evolved significantly. This evolution brings new complexity, new performance enhancements, and even new frameworks such as MLIR – changing the calculus of its performance characteristics. This motivated later research to re-examine these performance characteristics of compiler frameworks.

At the 2024 European LLVM Developers' Meeting, Mehdi Amini and Jeff Nui presented their keynote talk "How Slow is MLIR?" [30]. This talk aimed to quantify the feeling in the LLVM community that MLIR incurred a significant performance cost over LLVM alone, and produce metrics against which MLIR could be optimised. The presenters first discuss the implementation details of MLIR, including the design choices made to match common workloads. Next, they provide a set of micro-benchmarks for key functionality provided by MLIR, along with traditional benchmarks for constant folding and loop unrolling workloads. For the micro-benchmarks and constant folding workloads, MLIR is approximately four times slower than traditional LLVM. However, MLIR's more expressive IR representation yields an eighty-eight times speed up over LLVM for loop unrolling.

In addition to performance measurements of the LLVM and MLIR frameworks specifically, there is a body of research examining the performance of compilers more generally. In their 2024 paper, Engelke and Schwarz compare compile-times across frameworks for query compilation [47], finding Cranelift's [48] non-pointer chasing data structures yield a $20 - 35\%$ speedup over LLVM, supporting our hypothesis on the performance impact of dynamism. Engelke's earlier work also touches on compiler performance, proposing low-overhead approaches to binary instrumentation [49], rewriting at runtime [50], and JIT compilation [51] [52]. However, this body of research is much smaller in volume than the literature examining the performance of compiler generated code, despite its critical importance for developer productivity.

Our work differentiates itself from this existing research in two ways. Firstly, the above work focusses only on the performance characteristics of popular frameworks including LLVM, MLIR, and Cranelift. We extend this to also examine xDSL's performance. Secondly, having performance measurements and instrumentation for both MLIR and xDSL, we further extend the domain by contrasting the two frameworks through the lens of dynamism and its impact on performance.

8.2 Python language performance

Driven by Python's immense popularity, significant research effort has been expended developing tools and techniques to characterise its performance. This section discusses a relevant subset of these approaches, and contrasts them with our novel contributions.

8.2.1 Measuring application performance in Python

Reliable and accurate performance measurement is notoriously difficult. As such, its careful execution constitutes the main contribution of systems papers and theses [53] [54]. This difficulty comes from both sides of the hardware-software interface. For example, hardware optimisations such as hierarchical caches, branch predictors, and power management schemes exhibit complex emergent behaviour [55], making performance measurements less predictable and consistent. Similar confounding effects come from software, from process scheduling in the operating system to garbage collection in language runtimes [56]. Beyond this, advanced interpreters leverage runtime performance information for adaptive specialisation and JIT compilation, further muddling measurements. This phenomenon is explored by Barrett et al.'s "Virtual Machine Warmup Blows Hot and Cold" [29], where interpreter virtual machine warmup is shown to be highly variable, with benchmarks taking over 2000 iterations to reach a steady state. As such, accurate measurement of the performance characteristics of a Python program is more involved than the naïve approach of taking the wall time it takes to execute – requiring additional tools and techniques to guarantee reliable results. Fortunately, Python's strong ecosystem provides a wide variety of tools to achieve this goal, from the simple standard library `timeit` utility [33] to the `pyperf` package [34], with more complex control over confounding effects such as warm-ups and CPU isolation. Our work leverages these tools to make accurate measurements of compiler framework performance.

A key contribution of our research is our application of these tools to produce robust performance measurements and analysis of the xDSL user-extensible compiler frameworks, extending and contrasting similar work for MLIR. In addition to the research contribution of these measurements themselves, our work further supports ongoing research using the xDSL framework by providing re-usable performance benchmarks and associated tooling to measure performance. However, sometimes measurements with finer than end-to-end granularity are required. As such, our tooling also provides a simple user interface for applying performance profilers to these benchmarks.

8.2.2 Profiling to understand Python's performance

Existing profilers for Python typically operate at the function level. For example, Python's standard library provides the `profile` module, a Python-native tracing profiler, along with `cProfile`, a more performant C implementation of the same functionality [57]. These instrument each call event, providing accurate profiling information for each evaluated function. Beyond the standard library, profilers such as `pyinstrument` use statistical sampling rather than tracing to reduce overhead incurred by performance measurement [58]. In addition to this, the recent OSDI best paper winner "Triangulating Python Performance Issues with SCALENE" [59] introduces another profiler which focusses on the FFI boundary between C and Python, a key bottleneck for the best practice of delegating computation to fast low-level implementations. This delegation is particularly effective for structured workloads such as linear algebra, but is less suitable for highly dynamic workloads. Furthermore, profiling information at a finer granularity than the function level is often needed to deeply the performance of a program. `line_profiler` provides this functionality to a line level [60], but this is still one level of abstraction over the increasingly complex implementation of CPython's interpreter.

We fill this gap in the existing provision with ByteSight, a Python-native tracing performance profiler at the bytecode level. ByteSight extends existing work outputting and rewriting bytecode sequences [61] [62] [63], providing an easily installable package with the novel capability of performance profiling individual bytecode instructions. This contribution also unblocks other work in this thesis, facilitating close examination of specialised implementations and providing information about the performance of individual dynamic bytecode instructions.

8.3 Dynamism in programming languages

One common mechanism providing dynamism in programming languages is dynamic typing. This refers to programming languages where type-checking is performed at runtime, and variables can change type during the course of execution. In their essay "The next 7000 programming languages" [64], Chatley et al. discuss how the landscape of programming languages has changed since Landin's seminal 1966 paper "The next 700 programming languages" [65]. At the time of Landin's paper, there was already a split between dynamically typed languages such as Lisp and statically languages such as C and Algol. Lisp's runtime type checks incurred performance overhead and unexpected runtime type errors, but provided much greater expressivity and hence more productive development than static languages of the time. These trade-offs between static and dynamic languages remain much the same today, with Chatley et al. arguing that dynamically typed languages' expressivity results in "excellent library support", as they are better equipped to express structured data without a fixed schema.

Beyond dynamic typing, there are a wide variety of other mechanisms by which programming languages can provide dynamism. One mechanism is runtime meta-programming, which refers to code which can introspect and manipulate its own behaviour at runtime. An example of this is monkey-patching in Python, which allows the programmatic modification of objects at runtime [66]. Another mechanism is late binding, which refers to resolving method calls at runtime when they are invoked, as opposed to being statically linked ahead of time. Interestingly, ahead-of-time compiled languages typically considered static such as C++ provide this dynamic behaviour in the case of polymorphism. When a method is invoked on an object in an inheritance hierarchy, the correct implementation to execute is resolved at runtime using C++'s vtable mechanism [67]. Driesen and Hölzle quantify the performance overhead of this process in their paper "The Direct Cost of Virtual Function Calls in C++" [40], measuring both a direct cost as a result of the vtable indirection, and an indirect cost from precluded optimisation opportunities. This shows that the dynamic aspects of languages influence the details of their implementation, and the degree to which they can be optimised.

Ahead-of-time compilers rely on static mechanisms such as data-flow analysis to find valid optimisations. As they are run ahead of time, these static analyses have less information to reason about the dynamic runtime behaviour. In this dynamic case, some traditional optimisations cannot be guaranteed to be correct, and hence cannot be leveraged to improve program performance. For example, Hölze and Ungar's paper "Optimizing dynamically-dispatched calls with run-time type feedback" [68] argues that a function which is dynamically dispatched at runtime cannot be optimised through the code motion optimisation of function inlining, as the implementation which will be invoked is not known ahead of time. The authors then aim to address this with an experimental compiler implementing "type feedback", a profile-guided optimisation which inlines dynamically dispatched calls in object-orientated languages. This mirrors later work on instruction specialisation by Williams et al. [14] which lead to Python's specialising adaptive interpreter [37]. However, such approaches are specific to individual aspects of the language runtime, and being ahead of time can only optimise for a prediction of the runtime behaviour. Furthermore, runtime behaviour may differ significantly across inputs for some workloads, making this prediction less representative of real-world behaviour.

Our work applies this body of research to understand the impact of dynamic language features on the performance of dynamic workloads.

Chapter 9

Conclusion

We present performance measurements for the implementation of user-extensible compiler frameworks in dynamic languages through the proxy of xDSL, and contrast this with a replication of existing work measuring the performance of MLIR as a proxy for static language implementations. These measurements find a 110× slowdown for dynamic languages on real-world workloads, but capture the implementation details of the proxies in addition to the language runtimes (chapter 3). To address this, we present specialised versions of micro-benchmarks and a pattern rewriting workload, which. This specialisation process reduces the slowdown between frameworks to 12× (chapter 5). This effort revealed a gap in the tooling provision for highly granular profiling of Python code, leading to our development of ByteSight, a native tracing performance profiler for Python bytecode (chapter 4). ByteSight facilitates developing specialised implementations, and provides insights into the performance impact of dynamism on individual bytecode instructions. Using this tool, we examine optimisation techniques for dynamic language runtimes such as instruction specialisation and JIT compilation, comparing their impact between highly dynamic compiler frameworks with Python's representative suite of real-world workloads. We find that dynamic workloads exacerbate the impact of these optimisations, with instruction specialisation yielding a 10% greater speedup of dynamic over static workloads. This further reduces the slowdown between frameworks to 10× (chapter 6). Finally, we quantify the impact of dynamism on the performance of static and dynamic languages, elucidating the mechanism by which dynamic workloads preclude optimisations and incur overhead and justifying our earlier measurements (chapter 7).

Our work identifies dynamism in user-extensible compiler infrastructures as a result of the heterogeneous data structures used to represent IR, whose structure and contents is known only at runtime. It then quantifies the performance cost this dynamism incurs in their ahead-of-time compiled implementation. We contrast this with implementations in dynamic languages, empirically demonstrating that the performance overhead typically associated with such languages is lessened by both the workload's dynamic nature, and modern optimisation approaches to dynamic language runtimes. This contribution challenges the status quo of implementing user-extensible compiler frameworks in static, ahead-of-time compiled languages, typified by LLVM's MLIR in C++. Instead, we motivate the use of dynamic languages for these frameworks, showing that implementations such as xDSL which follow this approach present a desirable balance of framework performance with the developer productivity required to deliver modern workloads without delay.

Bibliography

[1] Richard M. Stallman and GCC Developer Community. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Scotts Valley, CA: CreateSpace, Feb. 2009. ISBN: 978-1-4414-1276-8.

[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*. International Symposium on Code Generation and Optimization, 2004. CGO 2004. Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. URL: https://ieeexplore.ieee.org/abstract/document/1281665 (visited on 11/18/2024).

[3] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1, 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://dl.acm.org/doi/10.1145/115372.115320 (visited on 06/09/2025).

[4] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Feb. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308. URL: https://ieeexplore.ieee.org/abstract/document/9370308 (visited on 04/11/2025).

[5] Mathieu Fehr et al. "xDSL: Sidekick Compilation for SSA-Based Compilers". In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '25. New York, NY, USA: Association for Computing Machinery, Mar. 1, 2025, pp. 179–192. ISBN: 979-8-4007-1275-3. DOI: 10.1145/3696443.3708945. URL: https://dl.acm.org/doi/10.1145/3696443.3708945 (visited on 04/11/2025).

[6] Hadi Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *IEEE Micro* 32.3 (May 2012), pp. 122–134. ISSN: 1937-4143. DOI: 10.1109/MM.2012.17. URL: https://ieeexplore.ieee.org/document/6175879 (visited on 10/14/2024).

[7] Adi Fuchs and David Wentzlaff. "The Accelerator Wall: Limits of Chip Specialization". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019 IEEE International Symposium on High Performance Com-

puter Architecture (HPCA). Feb. 2019, pp. 1–14. DOI: 10.1109/HPCA.2019.00023. URL: https://ieeexplore.ieee.org/document/8675237 (visited on 10/14/2024).

[8] Radosvet Desislavov, Fernando Martínez-Plumed, and José Hernández-Orallo. "Trends in AI Inference Energy Consumption: Beyond the Performance-vs-Parameter Laws of Deep Learning". In: *Sustainable Computing: Informatics and Systems* 38 (Apr. 1, 2023), p. 100857. ISSN: 2210-5379. DOI: 10.1016/j.suscom.2023.100857. URL: https://www.sciencedirect.com/science/article/pii/S2210537923000124 (visited on 06/07/2025).

[9] Sean Silva. "A High-Velocity Architecture for MLIR AI Compilers". CASCADE Poster Social (+ Compiler Tech Talk) (William Gates Building, University of Cambridge). Mar. 11, 2025. URL: https://talks.cam.ac.uk/talk/index/228928 (visited on 06/09/2025).

[10] DONALD E. Knuth and LUIS TRABB Pardo. "The Early Development of Programming Languages*†". In: *A History of Computing in the Twentieth Century*. Ed. by N. Metropolis, J. Howlett, and GIAN-CARLO Rota. San Diego: Academic Press, Jan. 1, 1980, pp. 197–273. ISBN: 978-0-12-491650-0. DOI: 10.1016/B978-0-12-491650-0.50019-8. URL: https://www.sciencedirect.com/science/article/pii/B9780124916500500198 (visited on 06/07/2025).

[11] Robert P. Wilson et al. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers". In: *SIGPLAN Not.* 29.12 (Dec. 1, 1994), pp. 31–37. ISSN: 0362-1340. DOI: 10.1145/193209.193217. URL: https://dl.acm.org/doi/10.1145/193209.193217 (visited on 06/09/2025).

[12] MLIR Team. *Traits - MLIR*. MLIR Code Documentation. URL: https://mlir.llvm.org/docs/Traits/ (visited on 06/11/2025).

[13] Mathieu Fehr et al. "IRDL: An IR Definition Language for SSA Compilers". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, June 9, 2022, pp. 199–212. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523700. URL: https://dl.acm.org/doi/10.1145/3519939.3523700 (visited on 04/11/2025).

[14] Kevin Williams, Jason McCandless, and David Gregg. "Dynamic Interpretation for Dynamic Scripting Languages". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. New York, NY, USA: Association for Computing Machinery, Apr. 24, 2010, pp. 278–287. ISBN: 978-1-60558-635-9. DOI: 10.1145/1772954.1772993. URL: https://dl.acm.org/doi/10.1145/1772954.1772993 (visited on 05/10/2025).

[15] Emery Berger. ""Python Performance Matters" by Emery Berger (Strange Loop 2022)" (Strange Loop Conference). Oct. 6, 2022. URL: https://www.youtube.com/watch?v=vVUnCXKuNOg (visited on 10/21/2024).

[16] Daniel Wang. “Evaluating Synchronization Overhead for Emerging Pointer Chasing Workloads”. Master of Science (MSc) in Information Technology. Uppsala, Sweden: Uppsala Universitet, 2025. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-550341 (visited on 06/10/2025).

[17] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Commun. ACM* 52.4 (Apr. 1, 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://dl.acm.org/doi/10.1145/1498765.1498785 (visited on 06/11/2025).

[18] Guido van Rossum. *Python/Cpython*. Python, Apr. 28, 2025. URL: https://github.com/python/cpython (visited on 04/28/2025).

[19] The PyPy Team. *Pypy/Pypy*. PyPy, June 6, 2025. URL: https://github.com/pypy/pypy (visited on 06/08/2025).

[20] Collin Winter and Jeffrey Yasskin. *Python/Pyperformance*. Python, May 6, 2025. URL: https://github.com/python/pyperformance (visited on 05/15/2025).

[21] John Aycock. “A Brief History of Just-in-Time”. In: *ACM Comput. Surv.* 35.2 (June 1, 2003), pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077. URL: https://dl.acm.org/doi/10.1145/857076.857077 (visited on 05/11/2025).

[22] Haoran Xu and Fredrik Kjolstad. “Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode”. In: *Artifact for Paper "Copy-and-Patch Compilation: A fast compilation algorithm for high-level languages and bytecode"* 5 (OOPSLA Oct. 15, 2021), 136:1–136:30. DOI: 10.1145/3485513. URL: https://dl.acm.org/doi/10.1145/3485513 (visited on 06/07/2025).

[23] Vaivaswatha Nagaraj. *Vaivaswatha/Pliron*. May 16, 2025. URL: https://github.com/vaivaswatha/pliron (visited on 05/17/2025).

[24] Eric Stoltz, Michael Wolfe, and Michael P. Gerlek. “Constant Propagation: A Fresh, Demand-Driven Look”. In: *Proceedings of the 1994 ACM Symposium on Applied Computing*. SAC ’94. New York, NY, USA: Association for Computing Machinery, Apr. 6, 1994, pp. 400–404. ISBN: 978-0-89791-647-9. DOI: 10.1145/326619.326791. URL: https://dl.acm.org/doi/10.1145/326619.326791 (visited on 05/16/2025).

[25] Michael Droettboom and Pauli Virtanen. *Airspeed-Velocity/Asv*. Airspeed Velocity, Apr. 27, 2025. URL: https://github.com/airspeed-velocity/asv (visited on 04/28/2025).

[26] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). New York, NY, USA: Association for Computing Machinery, Apr. 18, 1967, pp. 483–485. ISBN: 978-1-4503-7895-6. DOI: 10.1145/1465482.1465560. URL: https://dl.acm.org/doi/10.1145/1465482.1465560 (visited on 01/13/2025).

[27] Amit Sabne. “XLA: Compiling Machine Learning for Peak Performance”. In: *Google Research* (2020). URL: https://openxla.org/.

[28] R.H. Saavedra and A.J. Smith. "Performance Characterization of Optimizing Compilers". In: *IEEE Transactions on Software Engineering* 21.7 (July 1995), pp. 615–628. ISSN: 1939-3520. DOI: 10.1109/32.392982. URL: https://ieeexplore.ieee.org/document/392982/ (visited on 04/24/2025).

[29] Edd Barrett et al. "Virtual Machine Warmup Blows Hot and Cold". In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 12, 2017), pp. 1–27. ISSN: 2475-1421. DOI: 10.1145/3133876. arXiv: 1602.00602 [cs]. URL: http://arxiv.org/abs/1602.00602 (visited on 05/14/2025).

[30] Mehdi Amini and Jeff Nui. "How Slow Is MLIR?" 2024 European LLVM Developers' Meeting (Vienna, Austria). Apr. 2024. URL: https://www.youtube.com/watch?v=7qvVMUSxqz4.

[31] Victor Skvortsov. *Python behind the Scenes #4: How Python Bytecode Is Executed*. Oct. 30, 2020. URL: https://tenthousandmeters.com/blog/python-behind-the-scenes-4-how-python-bytecode-is-executed/ (visited on 05/27/2025).

[32] Apple Inc. *Mach_absolute_time*. Apple Developer Documentation. URL: https://developer.apple.com/documentation/kernel/1462446-mach_absolute_time (visited on 05/28/2025).

[33] Python Software Foundation. *Timeit — Measure Execution Time of Small Code Snippets*. Python documentation. URL: https://docs.python.org/3/library/timeit.html (visited on 05/28/2025).

[34] Victor Stinner. *Psf/Pyperf*. Python Software Foundation, May 12, 2025. URL: https://github.com/psf/pyperf (visited on 05/15/2025).

[35] Python Software Foundation. *3.3.2.4. Data Model, __slots__*. Python documentation. URL: https://docs.python.org/3/reference/datamodel.html#slots (visited on 06/07/2025).

[36] LLVM Project. *LLVM's Analysis and Transform Passes*. LLVM 21.0.0git documentation. URL: https://llvm.org/docs/Passes.html#memdep-memory-dependence-analysis (visited on 06/11/2025).

[37] Mark Shannon. *Specializing Adaptive Interpreter*. PEP 659. 2021. URL: https://peps.python.org/pep-0659/.

[38] Mark Shannon. "The Construction of High-Performance Virtual Machines for Dynamic Languages". PhD thesis. University of Glasgow, 2011. URL: https://eleanor.lib.gla.ac.uk/record=b2890492 (visited on 06/01/2025).

[39] Josh Haberman. *A Tail Calling Interpreter For Python (And Other Updates)*. Oct. 2, 2025. URL: https://blog.reverberate.org/2025/02/10/tail-call-updates.html (visited on 05/14/2025).

[40] Karel Driesen and Urs Hölzle. "The Direct Cost of Virtual Function Calls in C++". In: *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, Oct. 1, 1996, pp. 306–323. ISBN: 978-0-89791-

788-9. DOI: 10.1145/236337.236369. URL: https://dl.acm.org/doi/10.1145/236337.236369 (visited on 06/05/2025).

[41] Bjarne Stroustrup. "A History of C++: 1979–1991". In: *History of Programming Languages—II*. New York, NY, USA: Association for Computing Machinery, Jan. 1, 1996, pp. 699–769. ISBN: 978-0-201-89502-5. URL: https://doi.org/10.1145/234286.1057836 (visited on 06/04/2025).

[42] International Organization for Standardization. *ISO/IEC 14882:1998 Programming Languages — C++*. 1998. URL: https://www.iso.org/standard/25845.html (visited on 06/04/2025).

[43] Lois Goldthwaite. "Technical Report on C++ Performance". In: *ISO/IEC PDTR* 18015 (2006).

[44] MLIR Team. *MLIR: Include/Mlir/Support/TypeID.h Source File*. MLIR Code Documentation. URL: https://mlir.llvm.org/doxygen/TypeID_8h_source.html (visited on 06/11/2025).

[45] Nevena Milojkovic, Mohammad Ghafari, and Oscar Nierstrasz. "It's Duck (Typing) Season!" In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). May 2017, pp. 312–315. DOI: 10.1109/ICPC.2017.10. URL: https://ieeexplore.ieee.org/abstract/document/7961528 (visited on 06/05/2025).

[46] Bastien Abadie and Sylvestre Ledru. *Engineering Code Quality in the Firefox Browser: A Look at Our Tools and Challenges*. Mozilla Hacks – the Web developer blog. URL: https://hacks.mozilla.org/2020/04/code-quality-tools-at-mozilla (visited on 05/13/2025).

[47] Alexis Engelke and Tobias Schwarz. "Compile-Time Analysis of Compiler Frameworks for Query Compilation". In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Mar. 2024, pp. 233–244. DOI: 10.1109/CGO57630.2024.10444856. URL: https://ieeexplore.ieee.org/abstract/document/10444856 (visited on 06/10/2025).

[48] Bytecode Alliance. *Cranelift*. URL: https://cranelift.dev/ (visited on 06/10/2025).

[49] Alexis Engelke and Martin Schulz. "Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. New York, NY, USA: Association for Computing Machinery, Mar. 17, 2020, pp. 172–184. ISBN: 978-1-4503-7554-2. DOI: 10.1145/3381052.3381319. URL: https://dl.acm.org/doi/10.1145/3381052.3381319 (visited on 06/10/2025).

[50] Alexis Engelke and Josef Weidendorfer. "Using LLVM for Optimized Lightweight Binary Re-Writing at Runtime". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2017 IEEE International Parallel

and Distributed Processing Symposium Workshops (IPDPSW). May 2017, pp. 785–794. DOI: 10.1109/IPDPSW.2017.103. URL: https://ieeexplore.ieee.org/abstract/document/7965122 (visited on 06/10/2025).

[51] Tobias Schwarz, Tobias Kamm, and Alexis Engelke. *TPDE: A Fast Adaptable Compiler Back-End Framework*. arXiv.org. May 28, 2025. URL: https://arxiv.org/abs/2505.22610v1 (visited on 06/10/2025).

[52] Florian Drescher and Alexis Engelke. "Fast Template-Based Code Generation for MLIR". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2024, pp. 1–12. ISBN: 979-8-4007-0507-6. DOI: 10.1145/3640537.3641567. URL: https://dl.acm.org/doi/10.1145/3640537.3641567 (visited on 06/10/2025).

[53] Arthur Crapé. "Performance Analysis and Benchmarking of Python". Master's dissertation. Ghent, Belgium: Ghent University, 2020. 105 pp.

[54] David Harris-Birtill and Rose Harris-Birtill. "Understanding Computation Time: A Critical Discussion of Time as a Computational Performance Metric". In: *Time in Variance*. Brill, 2021, pp. 220–248.

[55] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2012. 858 pp. ISBN: 978-0-12-383872-8.

[56] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. "Myths and Realities: The Performance Impact of Garbage Collection". In: *SIGMETRICS Perform. Eval. Rev.* 32.1 (June 1, 2004), pp. 25–36. ISSN: 0163-5999. DOI: 10.1145/1012888.1005693. URL: https://dl.acm.org/doi/10.1145/1012888.1005693 (visited on 06/10/2025).

[57] Python Software Foundation. *The Python Profilers*. Python documentation. URL: https://docs.python.org/3/library/profile.html (visited on 05/28/2025).

[58] Joe Rickerby. *Pyinstrument*. May 28, 2025. URL: https://github.com/joerick/pyinstrument (visited on 05/28/2025).

[59] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. "Triangulating Python Performance Issues with SCALENE". In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 2023, pp. 51–64. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/berger (visited on 05/14/2025).

[60] Robert Kern and Jon Crall. *Pyutils/Line_profiler*. OpenPyUtils, May 28, 2025. URL: https://github.com/pyutils/line_profiler (visited on 05/28/2025).

[61] 0xec. *Coding — Reversing: Hacking the CPython Virtual Machine to Support Bytecode Debugging*. Coding — Reversing. Mar. 16, 2017. URL: https://0xec.blogspot.com/2017/03/hacking-cpython-virtual-machine-to.html (visited on 05/17/2025).

[62] Clement Rouault. *Understanding Python Execution from inside: A Python Assembly Tracer - Hakril's Blog*. URL: https://blog.hakril.net/articles/2-understanding-python-execution-tracer.html (visited on 05/17/2025).

[63] Ned Batchelder. *Wicked Hack: Python Bytecode Tracing*. Apr. 11, 2008. URL: https://nedbatchelder.com/blog/200804/wicked_hack_python_bytecode_tracing.html (visited on 05/17/2025).

[64] Robert Chatley, Alastair Donaldson, and Alan Mycroft. "The Next 7000 Programming Languages". In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Cham: Springer International Publishing, 2019, pp. 250–282. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_15. URL: https://doi.org/10.1007/978-3-319-91908-9_15 (visited on 05/10/2025).

[65] P. J. Landin. "The next 700 Programming Languages". In: *Commun. ACM* 9.3 (Mar. 1, 1966), pp. 157–166. ISSN: 0001-0782. DOI: 10.1145/365230.365257. URL: https://dl.acm.org/doi/10.1145/365230.365257 (visited on 05/10/2025).

[66] John Hunt. "Monkey Patching". In: *A Beginners Guide to Python 3 Programming*. Cham: Springer International Publishing, 2023, pp. 487–490. ISBN: 978-3-031-35122-8. DOI: 10.1007/978-3-031-35122-8_43. URL: https://doi.org/10.1007/978-3-031-35122-8_43.

[67] Guihao Liang. *Understand C++ Vtable from Assembly Code (Part 1)*. URL: https://guihao-liang.github.io/2020/05/30/what-is-vtable-in-cpp (visited on 06/11/2025).

[68] Urs Hölzle and David Ungar. "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback". In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI '94. New York, NY, USA: Association for Computing Machinery, June 1, 1994, pp. 326–336. ISBN: 978-0-89791-662-2. DOI: 10.1145/178243.178478. URL: https://dl.acm.org/doi/10.1145/178243.178478 (visited on 05/12/2025).

Appendix A

PyPerformance version comparison

The following section describes the procedure and provides the raw results used to evaluate the speed-ups between CPython versions. Each of the CPython versions was compiled from source with the `--enable-optimizations` and `--with-lto` configuration flags. This compilation was performed on the experimental machine (detailed in subsection 3.1.1) to avoid issues with cross-compilation. The PyPerformance tool runs a suite of benchmarks, each resulting in their own speed-up value (tabulated for each version comparison in Tables A.1, A.2, and A.3). These speed-up values are aggregated into a single speed-up value as the geometric mean. A geometric mean is used because it is less skewed by outlying data. The calculated speed-ups are shown in the main body of the thesis (Table 6.1).

```
py310.json
==========

Performance version: 1.11.0
Report on Linux-6.8.0-1029-aws-x86_64-with-glibc2.39
Number of logical CPUs: 16
Start date: 2025-06-02 23:12:50.096047
End date: 2025-06-02 23:39:13.222073

py311.json
==========

Performance version: 1.11.0
Report on Linux-6.8.0-1029-aws-x86_64-with-glibc2.39
Number of logical CPUs: 16
Start date: 2025-06-02 22:42:35.309501
End date: 2025-06-02 23:09:34.936710

+-------------------------------+------------+------------+--------------+------------------------+
| Benchmark                     | py310.json | py311.json | Change       | Significance           |
+===============================+============+============+==============+========================+
| async_generators              | 496 ms     | 405 ms     | 1.23x faster | Significant (t=102.94) |
+-------------------------------+------------+------------+--------------+------------------------+
| async_tree_cpu_io_mixed       | 1.13 sec   | 959 ms     | 1.18x faster | Significant (t=24.21)  |
+-------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager              | 854 ms     | 607 ms     | 1.41x faster | Significant (t=28.47)  |
```

```
+-------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_cpu_io_mixed | 1.14 sec   | 959 ms     | 1.18x faster | Significant (t=25.07)  |
+-------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_io           | 1.98 sec   | 1.39 sec   | 1.42x faster | Significant (t=115.69) |
+-------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_memoization  | 989 ms     | 753 ms     | 1.31x faster | Significant (t=70.85)  |
+-------------------------------+------------+------------+--------------+------------------------+
| async_tree_io                 | 1.98 sec   | 1.39 sec   | 1.42x faster | Significant (t=126.72) |
+-------------------------------+------------+------------+--------------+------------------------+
| async_tree_memoization        | 989 ms     | 754 ms     | 1.31x faster | Significant (t=80.45)  |
+-------------------------------+------------+------------+--------------+------------------------+
| async_tree_none               | 848 ms     | 607 ms     | 1.40x faster | Significant (t=28.13)  |
+-------------------------------+------------+------------+--------------+------------------------+
| asyncio_tcp                   | 907 ms     | 775 ms     | 1.17x faster | Significant (t=6.00)   |
+-------------------------------+------------+------------+--------------+------------------------+
| asyncio_tcp_ssl               | 2.32 sec   | 3.63 sec   | 1.56x slower | Significant (t=-11.69) |
+-------------------------------+------------+------------+--------------+------------------------+
| asyncio_websockets            | 643 ms     | 633 ms     | 1.02x faster | Not significant        |
+-------------------------------+------------+------------+--------------+------------------------+
| bench_mp_pool                 | 14.4 ms    | 12.8 ms    | 1.12x faster | Significant (t=7.41)   |
+-------------------------------+------------+------------+--------------+------------------------+
| bench_thread_pool             | 1.78 ms    | 1.69 ms    | 1.06x faster | Significant (t=31.34)  |
+-------------------------------+------------+------------+--------------+------------------------+
| chameleon                     | 9.92 ms    | 7.88 ms    | 1.26x faster | Significant (t=53.29)  |
+-------------------------------+------------+------------+--------------+------------------------+
| chaos                         | 130 ms     | 81.1 ms    | 1.61x faster | Significant (t=95.28)  |
+-------------------------------+------------+------------+--------------+------------------------+
| comprehensions                | 29.5 us    | 25.5 us    | 1.16x faster | Significant (t=90.45)  |
+-------------------------------+------------+------------+--------------+------------------------+
| coroutines                    | 35.7 ms    | 28.8 ms    | 1.24x faster | Significant (t=119.60) |
+-------------------------------+------------+------------+--------------+------------------------+
| coverage                      | 89.0 ms    | 84.8 ms    | 1.05x faster | Significant (t=5.95)   |
+-------------------------------+------------+------------+--------------+------------------------+
| create_gc_cycles              | 1.28 ms    | 1.11 ms    | 1.15x faster | Significant (t=24.86)  |
+-------------------------------+------------+------------+--------------+------------------------+
| crypto_pyaes                  | 134 ms     | 85.5 ms    | 1.56x faster | Significant (t=202.03) |
+-------------------------------+------------+------------+--------------+------------------------+
| dask                          | 525 ms     | 454 ms     | 1.16x faster | Significant (t=21.98)  |
+-------------------------------+------------+------------+--------------+------------------------+
| deepcopy                      | 503 us     | 407 us     | 1.24x faster | Significant (t=59.44)  |
+-------------------------------+------------+------------+--------------+------------------------+
| deepcopy_memo                 | 60.3 us    | 45.5 us    | 1.32x faster | Significant (t=77.90)  |
+-------------------------------+------------+------------+--------------+------------------------+
| deepcopy_reduce               | 4.45 us    | 3.59 us    | 1.24x faster | Significant (t=78.19)  |
+-------------------------------+------------+------------+--------------+------------------------+
| deltablue                     | 8.47 ms    | 4.08 ms    | 2.08x faster | Significant (t=150.55) |
+-------------------------------+------------+------------+--------------+------------------------+
| django_template               | 49.9 ms    | 38.9 ms    | 1.28x faster | Significant (t=45.49)  |
+-------------------------------+------------+------------+--------------+------------------------+
| docutils                      | 3.32 sec   | 2.70 sec   | 1.23x faster | Significant (t=110.18) |
+-------------------------------+------------+------------+--------------+------------------------+
| dulwich_log                   | 95.2 ms    | 79.0 ms    | 1.20x faster | Significant (t=59.47)  |
+-------------------------------+------------+------------+--------------+------------------------+
| fannkuch                      | 542 ms     | 434 ms     | 1.25x faster | Significant (t=33.10)  |
+-------------------------------+------------+------------+--------------+------------------------+
| float                         | 126 ms     | 86.2 ms    | 1.47x faster | Significant (t=127.64) |
+-------------------------------+------------+------------+--------------+------------------------+
| gc_traversal                  | 3.60 ms    | 3.56 ms    | 1.01x faster | Not significant        |
+-------------------------------+------------+------------+--------------+------------------------+
| generators                    | 60.4 ms    | 55.7 ms    | 1.08x faster | Significant (t=29.61)  |
+-------------------------------+------------+------------+--------------+------------------------+
| genshi_text                   | 33.8 ms    | 25.6 ms    | 1.32x faster | Significant (t=85.96)  |
+-------------------------------+------------+------------+--------------+------------------------+
```

```
| genshi_xml                    | 70.6 ms    | 61.1 ms    | 1.16x faster | Significant (t=29.82)  |
+-------------------------------+------------+------------+--------------+------------------------+
| go                            | 259 ms     | 166 ms     | 1.56x faster | Significant (t=120.34) |
+-------------------------------+------------+------------+--------------+------------------------+
| hexiom                        | 10.5 ms    | 7.43 ms    | 1.41x faster | Significant (t=34.56)  |
+-------------------------------+------------+------------+--------------+------------------------+
| html5lib                      | 106 ms     | 81.4 ms    | 1.30x faster | Significant (t=21.53)  |
+-------------------------------+------------+------------+--------------+------------------------+
| json_dumps                    | 14.7 ms    | 13.6 ms    | 1.08x faster | Significant (t=21.42)  |
+-------------------------------+------------+------------+--------------+------------------------+
| json_loads                    | 30.2 us    | 28.4 us    | 1.06x faster | Significant (t=23.39)  |
+-------------------------------+------------+------------+--------------+------------------------+
| logging_format                | 11.0 us    | 7.91 us    | 1.39x faster | Significant (t=50.52)  |
+-------------------------------+------------+------------+--------------+------------------------+
| logging_silent                | 197 ns     | 123 ns     | 1.60x faster | Significant (t=64.16)  |
+-------------------------------+------------+------------+--------------+------------------------+
| logging_simple                | 9.95 us    | 7.10 us    | 1.40x faster | Significant (t=39.79)  |
+-------------------------------+------------+------------+--------------+------------------------+
| mako                          | 17.3 ms    | 11.7 ms    | 1.47x faster | Significant (t=116.59) |
+-------------------------------+------------+------------+--------------+------------------------+
| mdp                           | 3.51 sec   | 3.20 sec   | 1.10x faster | Significant (t=16.77)  |
+-------------------------------+------------+------------+--------------+------------------------+
| meteor_contest                | 120 ms     | 109 ms     | 1.11x faster | Significant (t=58.59)  |
+-------------------------------+------------+------------+--------------+------------------------+
| nbody                         | 150 ms     | 102 ms     | 1.47x faster | Significant (t=21.91)  |
+-------------------------------+------------+------------+--------------+------------------------+
| nqueens                       | 111 ms     | 94.7 ms    | 1.17x faster | Significant (t=28.95)  |
+-------------------------------+------------+------------+--------------+------------------------+
| pathlib                       | 29.5 ms    | 27.6 ms    | 1.07x faster | Significant (t=14.71)  |
+-------------------------------+------------+------------+--------------+------------------------+
| pickle                        | 11.8 us    | 11.8 us    | 1.00x faster | Not significant        |
+-------------------------------+------------+------------+--------------+------------------------+
| pickle_dict                   | 28.5 us    | 29.5 us    | 1.03x slower | Significant (t=-8.47)  |
+-------------------------------+------------+------------+--------------+------------------------+
| pickle_list                   | 4.17 us    | 4.16 us    | 1.00x faster | Not significant        |
+-------------------------------+------------+------------+--------------+------------------------+
| pickle_pure_python            | 507 us     | 364 us     | 1.39x faster | Significant (t=201.27) |
+-------------------------------+------------+------------+--------------+------------------------+
| pidigits                      | 215 ms     | 208 ms     | 1.04x faster | Significant (t=16.13)  |
+-------------------------------+------------+------------+--------------+------------------------+
| pprint_pformat                | 2.30 sec   | 1.70 sec   | 1.35x faster | Significant (t=44.81)  |
+-------------------------------+------------+------------+--------------+------------------------+
| pprint_safe_repr              | 1.12 sec   | 825 ms     | 1.35x faster | Significant (t=62.68)  |
+-------------------------------+------------+------------+--------------+------------------------+
| pyflate                       | 747 ms     | 463 ms     | 1.61x faster | Significant (t=105.36) |
+-------------------------------+------------+------------+--------------+------------------------+
| python_startup                | 11.0 ms    | 10.2 ms    | 1.08x faster | Significant (t=113.04) |
+-------------------------------+------------+------------+--------------+------------------------+
| python_startup_no_site        | 7.18 ms    | 7.62 ms    | 1.06x slower | Significant (t=-63.11) |
+-------------------------------+------------+------------+--------------+------------------------+
| raytrace                      | 543 ms     | 355 ms     | 1.53x faster | Significant (t=97.20)  |
+-------------------------------+------------+------------+--------------+------------------------+
| regex_compile                 | 204 ms     | 161 ms     | 1.27x faster | Significant (t=68.71)  |
+-------------------------------+------------+------------+--------------+------------------------+
| regex_dna                     | 208 ms     | 182 ms     | 1.14x faster | Significant (t=45.65)  |
+-------------------------------+------------+------------+--------------+------------------------+
| regex_effbot                  | 3.36 ms    | 3.23 ms    | 1.04x faster | Significant (t=7.84)   |
+-------------------------------+------------+------------+--------------+------------------------+
| regex_v8                      | 27.6 ms    | 24.7 ms    | 1.12x faster | Significant (t=51.00)  |
+-------------------------------+------------+------------+--------------+------------------------+
| richards                      | 85.1 ms    | 56.0 ms    | 1.52x faster | Significant (t=84.28)  |
+-------------------------------+------------+------------+--------------+------------------------+
| richards_super                | 104 ms     | 67.5 ms    | 1.54x faster | Significant (t=81.27)  |
```

```
+-------------------------------+------------+------------+--------------+------------------------+
| scimark_fft                   | 485 ms     | 387 ms     | 1.25x faster | Significant (t=102.78) |
+-------------------------------+------------+------------+--------------+------------------------+
| scimark_lu                    | 197 ms     | 141 ms     | 1.39x faster | Significant (t=71.19)  |
+-------------------------------+------------+------------+--------------+------------------------+
| scimark_monte_carlo           | 128 ms     | 77.1 ms    | 1.66x faster | Significant (t=127.37) |
+-------------------------------+------------+------------+--------------+------------------------+
| scimark_sor                   | 232 ms     | 140 ms     | 1.66x faster | Significant (t=112.82) |
+-------------------------------+------------+------------+--------------+------------------------+
| scimark_sparse_mat_mult       | 6.67 ms    | 4.90 ms    | 1.36x faster | Significant (t=30.12)  |
+-------------------------------+------------+------------+--------------+------------------------+
| spectral_norm                 | 164 ms     | 131 ms     | 1.25x faster | Significant (t=52.68)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sqlalchemy_declarative        | 161 ms     | 133 ms     | 1.21x faster | Significant (t=22.47)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sqlalchemy_imperative         | 24.4 ms    | 21.1 ms    | 1.16x faster | Significant (t=33.55)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sqlglot_normalize             | 158 ms     | 128 ms     | 1.23x faster | Significant (t=86.82)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sqlglot_optimize              | 75.3 ms    | 61.3 ms    | 1.23x faster | Significant (t=89.97)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sqlglot_parse                 | 2.33 ms    | 1.62 ms    | 1.44x faster | Significant (t=129.85) |
+-------------------------------+------------+------------+--------------+------------------------+
| sqlglot_transpile             | 2.76 ms    | 1.95 ms    | 1.41x faster | Significant (t=83.09)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sqlite_synth                  | 3.68 us    | 3.07 us    | 1.20x faster | Significant (t=76.70)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sympy_expand                  | 622 ms     | 537 ms     | 1.16x faster | Significant (t=29.76)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sympy_integrate               | 27.0 ms    | 22.1 ms    | 1.22x faster | Significant (t=52.24)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sympy_str                     | 373 ms     | 324 ms     | 1.15x faster | Significant (t=29.39)  |
+-------------------------------+------------+------------+--------------+------------------------+
| sympy_sum                     | 209 ms     | 181 ms     | 1.15x faster | Significant (t=43.71)  |
+-------------------------------+------------+------------+--------------+------------------------+
| telco                         | 8.34 ms    | 7.67 ms    | 1.09x faster | Significant (t=22.23)  |
+-------------------------------+------------+------------+--------------+------------------------+
| tomli_loads                   | 3.29 sec   | 2.50 sec   | 1.32x faster | Significant (t=66.74)  |
+-------------------------------+------------+------------+--------------+------------------------+
| tornado_http                  | 168 ms     | 137 ms     | 1.23x faster | Significant (t=27.00)  |
+-------------------------------+------------+------------+--------------+------------------------+
| typing_runtime_protocols      | 607 us     | 512 us     | 1.18x faster | Significant (t=30.73)  |
+-------------------------------+------------+------------+--------------+------------------------+
| unpack_sequence               | 58.4 ns    | 44.8 ns    | 1.30x faster | Significant (t=5.12)   |
+-------------------------------+------------+------------+--------------+------------------------+
| unpickle                      | 15.8 us    | 14.5 us    | 1.09x faster | Significant (t=18.21)  |
+-------------------------------+------------+------------+--------------+------------------------+
| unpickle_list                 | 5.39 us    | 5.20 us    | 1.04x faster | Significant (t=9.47)   |
+-------------------------------+------------+------------+--------------+------------------------+
| unpickle_pure_python          | 360 us     | 277 us     | 1.30x faster | Significant (t=79.85)  |
+-------------------------------+------------+------------+--------------+------------------------+
| xml_etree_generate            | 109 ms     | 91.8 ms    | 1.19x faster | Significant (t=70.86)  |
+-------------------------------+------------+------------+--------------+------------------------+
| xml_etree_iterparse           | 119 ms     | 110 ms     | 1.08x faster | Significant (t=29.01)  |
+-------------------------------+------------+------------+--------------+------------------------+
| xml_etree_parse               | 166 ms     | 165 ms     | 1.01x faster | Not significant        |
+-------------------------------+------------+------------+--------------+------------------------+
| xml_etree_process             | 87.5 ms    | 64.5 ms    | 1.36x faster | Significant (t=144.92) |
+-------------------------------+------------+------------+--------------+------------------------+

Skipped 8 benchmarks only in py311.json: async_tree_cpu_io_mixed_tg, async_tree_eager_cpu_io_mixed_tg, async_tree_eager_io_tg, async_tree_eager_memoization_tg, async_tree_eager_tg, async_tree_io_tg, async_tree_memoization_tg, async_tree_none_tg
```

Listing A.1: Comparison table of PyPerformance benchmark results between CPython versions 3.10.17 and 3.11.12.

```
py311.json
==========

Performance version: 1.11.0
Report on Linux-6.8.0-1029-aws-x86_64-with-glibc2.39
Number of logical CPUs: 16
Start date: 2025-06-02 22:42:35.309501
End date: 2025-06-02 23:09:34.936710

py313.json
==========

Performance version: 1.11.0
Report on Linux-6.8.0-1029-aws-x86_64-with-glibc2.39
Number of logical CPUs: 16
Start date: 2025-06-02 22:07:57.491156
End date: 2025-06-02 22:31:36.629262

+----------------------------------+------------+------------+--------------+------------------------+
| Benchmark                        | py311.json | py313.json | Change       | Significance           |
+==================================+============+============+==============+========================+
| async_generators                 | 405 ms     | 460 ms     | 1.14x slower | Significant (t=-27.58) |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_cpu_io_mixed          | 959 ms     | 677 ms     | 1.42x faster | Significant (t=106.37) |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_cpu_io_mixed_tg       | 829 ms     | 742 ms     | 1.12x faster | Significant (t=20.80)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager                 | 607 ms     | 135 ms     | 4.51x faster | Significant (t=596.51) |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_cpu_io_mixed    | 959 ms     | 468 ms     | 2.05x faster | Significant (t=134.71) |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_cpu_io_mixed_tg | 827 ms     | 627 ms     | 1.32x faster | Significant (t=36.13)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_io              | 1.39 sec   | 1.00 sec   | 1.39x faster | Significant (t=32.35)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_io_tg           | 1.30 sec   | 1.10 sec   | 1.19x faster | Significant (t=11.67)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_memoization     | 753 ms     | 281 ms     | 2.68x faster | Significant (t=301.57) |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_memoization_tg  | 694 ms     | 438 ms     | 1.58x faster | Significant (t=34.92)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_eager_tg              | 528 ms     | 318 ms     | 1.66x faster | Significant (t=56.23)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_io                    | 1.39 sec   | 966 ms     | 1.44x faster | Significant (t=36.45)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_io_tg                 | 1.30 sec   | 1.00 sec   | 1.30x faster | Significant (t=68.18)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_memoization           | 754 ms     | 536 ms     | 1.41x faster | Significant (t=16.34)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_memoization_tg        | 700 ms     | 533 ms     | 1.31x faster | Significant (t=24.40)  |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_none                  | 607 ms     | 412 ms     | 1.47x faster | Significant (t=144.13) |
+----------------------------------+------------+------------+--------------+------------------------+
| async_tree_none_tg               | 528 ms     | 381 ms     | 1.39x faster | Significant (t=105.89) |
+----------------------------------+------------+------------+--------------+------------------------+
| asyncio_tcp                      | 775 ms     | 476 ms     | 1.63x faster | Significant (t=124.10) |
+----------------------------------+------------+------------+--------------+------------------------+
| asyncio_tcp_ssl                  | 3.63 sec   | 1.74 sec   | 2.08x faster | Significant (t=853.90) |
+----------------------------------+------------+------------+--------------+------------------------+
```

asyncio_websockets	633 ms	642 ms	1.01x slower	Not significant
bench_mp_pool	12.8 ms	12.7 ms	1.01x faster	Not significant
bench_thread_pool	1.69 ms	1.68 ms	1.01x faster	Not significant
chameleon	7.88 ms	8.15 ms	1.03x slower	Significant (t=-11.58)
chaos	81.1 ms	71.5 ms	1.13x faster	Significant (t=23.56)
comprehensions	25.5 us	20.2 us	1.26x faster	Significant (t=38.57)
coroutines	28.8 ms	26.9 ms	1.07x faster	Significant (t=27.12)
coverage	84.8 ms	108 ms	1.27x slower	Significant (t=-31.89)
create_gc_cycles	1.11 ms	1.23 ms	1.11x slower	Significant (t=-21.36)
crypto_pyaes	85.5 ms	78.7 ms	1.09x faster	Significant (t=26.47)
dask	454 ms	449 ms	1.01x faster	Not significant
deepcopy	407 us	449 us	1.11x slower	Significant (t=-49.99)
deepcopy_memo	45.5 us	49.4 us	1.08x slower	Significant (t=-38.52)
deepcopy_reduce	3.59 us	4.05 us	1.13x slower	Significant (t=-29.35)
deltablue	4.08 ms	3.55 ms	1.15x faster	Significant (t=57.73)
django_template	38.9 ms	40.9 ms	1.05x slower	Significant (t=-8.99)
docutils	2.70 sec	2.74 sec	1.01x slower	Not significant
dulwich_log	79.0 ms	79.3 ms	1.00x slower	Not significant
fannkuch	434 ms	470 ms	1.08x slower	Significant (t=-11.43)
float	86.2 ms	95.1 ms	1.10x slower	Significant (t=-31.15)
gc_traversal	3.56 ms	3.75 ms	1.05x slower	Significant (t=-4.61)
generators	55.7 ms	36.9 ms	1.51x faster	Significant (t=122.22)
genshi_text	25.6 ms	26.3 ms	1.03x slower	Significant (t=-4.97)
genshi_xml	61.1 ms	61.2 ms	1.00x slower	Not significant
go	166 ms	171 ms	1.03x slower	Significant (t=-8.53)
hexiom	7.43 ms	6.89 ms	1.08x faster	Significant (t=25.05)
html5lib	81.4 ms	85.1 ms	1.05x slower	Significant (t=-4.97)
json_dumps	13.6 ms	12.0 ms	1.14x faster	Significant (t=44.50)
json_loads	28.4 us	30.3 us	1.07x slower	Significant (t=-23.47)
logging_format	7.91 us	7.79 us	1.02x faster	Not significant
logging_silent	123 ns	137 ns	1.11x slower	Significant (t=-14.30)
logging_simple	7.10 us	6.75 us	1.05x faster	Significant (t=7.92)

```
+----------------------------------+------------+------------+--------------+-------------------------+
| mako                             | 11.7 ms    | 12.8 ms    | 1.09x slower | Significant (t=-30.48)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| mdp                              | 3.20 sec   | 3.02 sec   | 1.06x faster | Significant (t=9.91)    |
+----------------------------------+------------+------------+--------------+-------------------------+
| meteor_contest                   | 109 ms     | 116 ms     | 1.07x slower | Significant (t=-19.41)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| nbody                            | 102 ms     | 106 ms     | 1.04x slower | Significant (t=-12.11)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| nqueens                          | 94.7 ms    | 92.6 ms    | 1.02x faster | Significant (t=4.58)    |
+----------------------------------+------------+------------+--------------+-------------------------+
| pathlib                          | 27.6 ms    | 26.0 ms    | 1.06x faster | Significant (t=13.29)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| pickle                           | 11.8 us    | 14.9 us    | 1.27x slower | Significant (t=-54.21)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| pickle_dict                      | 29.5 us    | 33.3 us    | 1.13x slower | Significant (t=-43.03)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| pickle_list                      | 4.16 us    | 4.98 us    | 1.20x slower | Significant (t=-39.29)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| pickle_pure_python               | 364 us     | 359 us     | 1.01x faster | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| pidigits                         | 208 ms     | 218 ms     | 1.05x slower | Significant (t=-34.21)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| pprint_pformat                   | 1.70 sec   | 1.82 sec   | 1.07x slower | Significant (t=-14.43)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| pprint_safe_repr                 | 825 ms     | 884 ms     | 1.07x slower | Significant (t=-22.24)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| pyflate                          | 463 ms     | 513 ms     | 1.11x slower | Significant (t=-25.73)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| python_startup                   | 10.2 ms    | 12.5 ms    | 1.22x slower | Significant (t=-292.44) |
+----------------------------------+------------+------------+--------------+-------------------------+
| python_startup_no_site           | 7.62 ms    | 8.57 ms    | 1.12x slower | Significant (t=-136.29) |
+----------------------------------+------------+------------+--------------+-------------------------+
| raytrace                         | 355 ms     | 309 ms     | 1.15x faster | Significant (t=43.55)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| regex_compile                    | 161 ms     | 156 ms     | 1.03x faster | Significant (t=10.62)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| regex_dna                        | 182 ms     | 215 ms     | 1.18x slower | Significant (t=-38.47)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| regex_effbot                     | 3.23 ms    | 3.24 ms    | 1.00x slower | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| regex_v8                         | 24.7 ms    | 27.9 ms    | 1.13x slower | Significant (t=-27.49)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| richards                         | 56.0 ms    | 60.7 ms    | 1.08x slower | Significant (t=-12.87)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| richards_super                   | 67.5 ms    | 67.5 ms    | 1.00x slower | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| scimark_fft                      | 387 ms     | 419 ms     | 1.08x slower | Significant (t=-18.47)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| scimark_lu                       | 141 ms     | 137 ms     | 1.03x faster | Significant (t=8.59)    |
+----------------------------------+------------+------------+--------------+-------------------------+
| scimark_monte_carlo              | 77.1 ms    | 78.9 ms    | 1.02x slower | Significant (t=-8.31)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| scimark_sor                      | 140 ms     | 157 ms     | 1.12x slower | Significant (t=-62.46)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| scimark_sparse_mat_mult          | 4.90 ms    | 5.50 ms    | 1.12x slower | Significant (t=-16.82)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| spectral_norm                    | 131 ms     | 128 ms     | 1.02x faster | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| sqlglot_normalize                | 128 ms     | 129 ms     | 1.01x slower | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| sqlglot_optimize                 | 61.3 ms    | 62.6 ms    | 1.02x slower | Significant (t=-6.14)   |
+----------------------------------+------------+------------+--------------+-------------------------+
```

```
| sqlglot_parse                    | 1.62 ms    | 1.49 ms    | 1.08x faster | Significant (t=43.70)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| sqlglot_transpile                | 1.95 ms    | 1.82 ms    | 1.07x faster | Significant (t=24.15)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| sqlite_synth                     | 3.07 us    | 3.37 us    | 1.10x slower | Significant (t=-34.84)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| sympy_expand                     | 537 ms     | 523 ms     | 1.03x faster | Significant (t=6.58)    |
+----------------------------------+------------+------------+--------------+-------------------------+
| sympy_integrate                  | 22.1 ms    | 21.2 ms    | 1.04x faster | Significant (t=18.00)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| sympy_str                        | 324 ms     | 307 ms     | 1.06x faster | Significant (t=13.82)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| sympy_sum                        | 181 ms     | 162 ms     | 1.12x faster | Significant (t=31.73)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| telco                            | 7.67 ms    | 9.23 ms    | 1.20x slower | Significant (t=-39.84)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| tomli_loads                      | 2.50 sec   | 2.50 sec   | 1.00x slower | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| tornado_http                     | 137 ms     | 134 ms     | 1.02x faster | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| typing_runtime_protocols         | 512 us     | 189 us     | 2.71x faster | Significant (t=186.48)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| unpack_sequence                  | 44.8 ns    | 52.9 ns    | 1.18x slower | Significant (t=-3.27)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| unpickle                         | 14.5 us    | 16.0 us    | 1.10x slower | Significant (t=-19.09)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| unpickle_list                    | 5.20 us    | 5.64 us    | 1.09x slower | Significant (t=-31.35)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| unpickle_pure_python             | 277 us     | 262 us     | 1.05x faster | Significant (t=18.66)   |
+----------------------------------+------------+------------+--------------+-------------------------+
| xml_etree_generate               | 91.8 ms    | 99.3 ms    | 1.08x slower | Significant (t=-39.71)  |
+----------------------------------+------------+------------+--------------+-------------------------+
| xml_etree_iterparse              | 110 ms     | 112 ms     | 1.02x slower | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| xml_etree_parse                  | 165 ms     | 164 ms     | 1.01x faster | Not significant         |
+----------------------------------+------------+------------+--------------+-------------------------+
| xml_etree_process                | 64.5 ms    | 68.7 ms    | 1.07x slower | Significant (t=-39.35)  |
+----------------------------------+------------+------------+--------------+-------------------------+

Skipped 2 benchmarks only in py311.json: sqlalchemy_declarative, sqlalchemy_imperative

Skipped 1 benchmarks only in py313.json: 2to3
```

Listing A.2: Comparison table of PyPerformance benchmark results between CPython versions 3.11.12 and 3.13.3.

```
py313.json
==========

Performance version: 1.11.0
Report on Linux-6.8.0-1029-aws-x86_64-with-glibc2.39
Number of logical CPUs: 16
Start date: 2025-06-02 22:07:57.491156
End date: 2025-06-02 22:31:36.629262

py313-jit.json
==============

Performance version: 1.11.0
Report on Linux-6.8.0-1029-aws-x86_64-with-glibc2.39
Number of logical CPUs: 16
Start date: 2025-06-02 21:40:29.802724
End date: 2025-06-02 22:04:08.722964

```

Benchmark	py313.json	py313-jit.json	Change	Significance
async_generators	460 ms	485 ms	1.05x slower	Significant (t=-8.31)
async_tree_cpu_io_mixed	677 ms	672 ms	1.01x faster	Not significant
async_tree_cpu_io_mixed_tg	742 ms	735 ms	1.01x faster	Not significant
async_tree_eager	135 ms	138 ms	1.03x slower	Significant (t=-5.82)
async_tree_eager_cpu_io_mixed	468 ms	465 ms	1.01x faster	Not significant
async_tree_eager_cpu_io_mixed_tg	627 ms	620 ms	1.01x faster	Not significant
async_tree_eager_io	1.00 sec	997 ms	1.00x faster	Not significant
async_tree_eager_io_tg	1.10 sec	1.12 sec	1.02x slower	Not significant
async_tree_eager_memoization	281 ms	289 ms	1.03x slower	Significant (t=-3.93)
async_tree_eager_memoization_tg	438 ms	439 ms	1.00x slower	Not significant
async_tree_eager_tg	318 ms	322 ms	1.01x slower	Not significant
async_tree_io	966 ms	960 ms	1.01x faster	Not significant
async_tree_io_tg	1.00 sec	990 ms	1.01x faster	Not significant
async_tree_memoization	536 ms	538 ms	1.00x slower	Not significant
async_tree_memoization_tg	533 ms	531 ms	1.00x faster	Not significant
async_tree_none	412 ms	414 ms	1.01x slower	Not significant
async_tree_none_tg	381 ms	383 ms	1.00x slower	Not significant
asyncio_tcp	476 ms	504 ms	1.06x slower	Significant (t=-10.55)
asyncio_tcp_ssl	1.74 sec	1.75 sec	1.00x slower	Not significant
asyncio_websockets	642 ms	647 ms	1.01x slower	Not significant
bench_mp_pool	12.7 ms	13.1 ms	1.03x slower	Not significant
bench_thread_pool	1.68 ms	1.74 ms	1.04x slower	Significant (t=-13.58)
chameleon	8.15 ms	8.22 ms	1.01x slower	Not significant
chaos	71.5 ms	71.2 ms	1.00x faster	Not significant
comprehensions	20.2 us	19.5 us	1.03x faster	Significant (t=4.31)
coroutines	26.9 ms	26.9 ms	1.00x faster	Not significant
coverage	108 ms	96.5 ms	1.12x faster	Significant (t=23.68)
create_gc_cycles	1.23 ms	1.24 ms	1.01x slower	Not significant
crypto_pyaes	78.7 ms	76.2 ms	1.03x faster	Significant (t=5.83)
dask	449 ms	455 ms	1.01x slower	Not significant

```
| deepcopy                         | 449 us     | 466 us         | 1.04x slower | Significant (t=-4.65)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| deepcopy_memo                    | 49.4 us    | 45.5 us        | 1.09x faster | Significant (t=40.04)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| deepcopy_reduce                  | 4.05 us    | 4.11 us        | 1.01x slower | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| deltablue                        | 3.55 ms    | 3.97 ms        | 1.12x slower | Significant (t=-55.41)  |
+----------------------------------+------------+----------------+--------------+-------------------------+
| django_template                  | 40.9 ms    | 43.3 ms        | 1.06x slower | Significant (t=-7.95)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| docutils                         | 2.74 sec   | 2.87 sec       | 1.05x slower | Significant (t=-8.39)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| dulwich_log                      | 79.3 ms    | 81.1 ms        | 1.02x slower | Significant (t=-2.44)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| fannkuch                         | 470 ms     | 423 ms         | 1.11x faster | Significant (t=24.08)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| float                            | 95.1 ms    | 82.4 ms        | 1.15x faster | Significant (t=37.45)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| gc_traversal                     | 3.75 ms    | 3.76 ms        | 1.00x slower | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| generators                       | 36.9 ms    | 37.1 ms        | 1.00x slower | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| genshi_text                      | 26.3 ms    | 30.2 ms        | 1.15x slower | Significant (t=-22.09)  |
+----------------------------------+------------+----------------+--------------+-------------------------+
| genshi_xml                       | 61.2 ms    | 68.5 ms        | 1.12x slower | Significant (t=-32.32)  |
+----------------------------------+------------+----------------+--------------+-------------------------+
| go                               | 171 ms     | 174 ms         | 1.02x slower | Significant (t=-5.09)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| hexiom                           | 6.89 ms    | 7.13 ms        | 1.04x slower | Significant (t=-8.44)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| html5lib                         | 85.1 ms    | 82.9 ms        | 1.03x faster | Significant (t=9.55)    |
+----------------------------------+------------+----------------+--------------+-------------------------+
| json_dumps                       | 12.0 ms    | 11.8 ms        | 1.01x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| json_loads                       | 30.3 us    | 30.4 us        | 1.00x slower | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| logging_format                   | 7.79 us    | 7.25 us        | 1.07x faster | Significant (t=7.00)    |
+----------------------------------+------------+----------------+--------------+-------------------------+
| logging_silent                   | 137 ns     | 135 ns         | 1.01x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| logging_simple                   | 6.75 us    | 6.65 us        | 1.02x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| mako                             | 12.8 ms    | 11.6 ms        | 1.10x faster | Significant (t=12.42)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| mdp                              | 3.02 sec   | 3.06 sec       | 1.01x slower | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| meteor_contest                   | 116 ms     | 117 ms         | 1.00x slower | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| nbody                            | 106 ms     | 113 ms         | 1.06x slower | Significant (t=-14.11)  |
+----------------------------------+------------+----------------+--------------+-------------------------+
| nqueens                          | 92.6 ms    | 104 ms         | 1.12x slower | Significant (t=-18.68)  |
+----------------------------------+------------+----------------+--------------+-------------------------+
| pathlib                          | 26.0 ms    | 26.1 ms        | 1.00x slower | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| pickle                           | 14.9 us    | 14.7 us        | 1.01x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| pickle_dict                      | 33.3 us    | 32.6 us        | 1.02x faster | Significant (t=4.89)    |
+----------------------------------+------------+----------------+--------------+-------------------------+
| pickle_list                      | 4.98 us    | 4.97 us        | 1.00x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| pickle_pure_python               | 359 us     | 367 us         | 1.02x slower | Significant (t=-10.93)  |
+----------------------------------+------------+----------------+--------------+-------------------------+
| pidigits                         | 218 ms     | 212 ms         | 1.03x faster | Significant (t=32.66)   |
```

pprint_pformat	1.82 sec	1.81 sec	1.00x faster	Not significant
pprint_safe_repr	884 ms	886 ms	1.00x slower	Not significant
pyflate	513 ms	482 ms	1.06x faster	Significant (t=13.39)
python_startup	12.5 ms	14.3 ms	1.15x slower	Significant (t=-169.42)
python_startup_no_site	8.57 ms	10.4 ms	1.22x slower	Significant (t=-303.79)
raytrace	309 ms	318 ms	1.03x slower	Significant (t=-9.91)
regex_compile	156 ms	153 ms	1.02x faster	Significant (t=7.49)
regex_dna	215 ms	199 ms	1.08x faster	Significant (t=20.71)
regex_effbot	3.24 ms	3.13 ms	1.04x faster	Significant (t=4.77)
regex_v8	27.9 ms	28.0 ms	1.00x slower	Not significant
richards	60.7 ms	47.6 ms	1.27x faster	Significant (t=46.32)
richards_super	67.5 ms	55.4 ms	1.22x faster	Significant (t=40.93)
scimark_fft	419 ms	359 ms	1.17x faster	Significant (t=32.96)
scimark_lu	137 ms	157 ms	1.14x slower	Significant (t=-46.95)
scimark_monte_carlo	78.9 ms	80.7 ms	1.02x slower	Significant (t=-7.75)
scimark_sor	157 ms	162 ms	1.03x slower	Significant (t=-15.13)
scimark_sparse_mat_mult	5.50 ms	4.64 ms	1.19x faster	Significant (t=23.43)
spectral_norm	128 ms	107 ms	1.20x faster	Significant (t=41.15)
sqlglot_normalize	129 ms	132 ms	1.03x slower	Significant (t=-4.35)
sqlglot_optimize	62.6 ms	65.2 ms	1.04x slower	Significant (t=-9.74)
sqlglot_parse	1.49 ms	1.46 ms	1.02x faster	Not significant
sqlglot_transpile	1.82 ms	1.81 ms	1.01x faster	Not significant
sqlite_synth	3.37 us	3.27 us	1.03x faster	Significant (t=8.77)
sympy_expand	523 ms	574 ms	1.10x slower	Significant (t=-19.84)
sympy_integrate	21.2 ms	22.8 ms	1.07x slower	Significant (t=-31.11)
sympy_str	307 ms	327 ms	1.06x slower	Significant (t=-16.63)
sympy_sum	162 ms	175 ms	1.08x slower	Significant (t=-22.14)
telco	9.23 ms	9.47 ms	1.03x slower	Significant (t=-6.64)
tomli_loads	2.50 sec	2.27 sec	1.10x faster	Significant (t=28.24)
tornado_http	134 ms	139 ms	1.04x slower	Significant (t=-4.64)
typing_runtime_protocols	189 us	199 us	1.05x slower	Significant (t=-6.47)

```
| unpack_sequence                  | 52.9 ns    | 197 ns         | 3.73x slower | Significant (t=-366.64) |
+----------------------------------+------------+----------------+--------------+-------------------------+
| unpickle                         | 16.0 us    | 15.9 us        | 1.01x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| unpickle_list                    | 5.64 us    | 5.36 us        | 1.05x faster | Significant (t=12.69)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| unpickle_pure_python             | 262 us     | 260 us         | 1.01x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| xml_etree_generate               | 99.3 ms    | 96.7 ms        | 1.03x faster | Significant (t=11.44)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| xml_etree_iterparse              | 112 ms     | 108 ms         | 1.03x faster | Significant (t=13.58)   |
+----------------------------------+------------+----------------+--------------+-------------------------+
| xml_etree_parse                  | 164 ms     | 164 ms         | 1.00x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+
| xml_etree_process                | 68.7 ms    | 68.6 ms        | 1.00x faster | Not significant         |
+----------------------------------+------------+----------------+--------------+-------------------------+

Skipped 1 benchmarks only in py313.json: 2to3
```

Listing A.3: Comparison table of PyPerformance benchmark results between CPython 3.13.3 with and without the JIT enabled.

Appendix B

MLIR workloads

B.1 Constant folding

```python
def constant_folding_module(size: int) -> ModuleOp:
    """Generate a constant folding workload of a given size.

    The output of running the command
    `print(WorkloadBuilder().constant_folding_module(size=5))` is shown
    below:

    ```mlir
    "builtin.module"() ({
        %0 = "arith.constant"() {"value" = 865 : i32} : () -> i32
        %1 = "arith.constant"() {"value" = 395 : i32} : () -> i32
        %2 = "arith.addi"(%1, %0) : (i32, i32) -> i32
        %3 = "arith.constant"() {"value" = 777 : i32} : () -> i32
        %4 = "arith.addi"(%3, %2) : (i32, i32) -> i32
        %5 = "arith.constant"() {"value" = 912 : i32} : () -> i32
        "test.op"(%4) : (i32) -> ()
    }) : () -> ()
    ```
    """
    assert size >= 0
    random.seed(RANDOM_SEED)
    ops: list[Operation] = []
    ops.append(ConstantOp(IntegerAttr(random.randint(1, 1000), i32)))
    for i in range(1, size + 1):
        if i % 2 == 0:
            ops.append(AddiOp(ops[i - 1], ops[i - 2]))
        else:
            ops.append(ConstantOp(IntegerAttr(random.randint(1, 1000), i32)))
    ops.append(TestOp([ops[(size // 2) * 2]]))
    return ModuleOp(ops)
```

Listing B.1: Python implementation of a parameterised generator for constant folding over integers in xDSL.

Appendix C

MLIR benchmark results

C.1 Pipeline phase micro-benchmark results

C.2 How Slow is MLIR micro-benchmark results

The following section describes the procedure and provides the raw results from “How Slow is MLIR’s” micro-benchmarks.

```bash
 git clone https://github.com/EdmundGoodman/llvm-project-benchmarks --depth 1
 mkdir -p llvm-project-benchmarks/build/
 cd llvm-project-benchmarks/build
 cmake -G Ninja ../llvm \
    -DLLVM_ENABLE_PROJECTS=mlir \
    -DLLVM_TARGETS_TO_BUILD="host" \
    -DLLVM_ENABLE_BENCHMARKS=ON \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_C_COMPILER=clang-18 -DCMAKE_CXX_COMPILER=clang++-18
./tools/mlir/unittests/Benchmarks/MLIR_IR_Benchmark
```

Listing C.1: Bash commands to downloads, compiler, and run the benchmarks from “How Slow is MLIR”.

```
2025-05-14T12:19:17+00:00
Running ./tools/mlir/unittests/Benchmarks/MLIR_IR_Benchmark
Run on (16 X 2799.99 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 512 KiB (x8)
  L3 Unified 16384 KiB (x2)
Load Average: 0.02, 0.06, 0.02
-----------------------------------------------------------------------------------------------------
Benchmark                                                                  Time             CPU   Iterations
-----------------------------------------------------------------------------------------------------
Analysis/symbolTable/10                                                  258 ns          258 ns      2695939
Analysis/symbolTable/64                                                 1280 ns         1280 ns       543255
```

```
Analysis/symbolTable/512                               11790 ns      11788 ns      60690
Analysis/symbolTable/4096                             224244 ns     224225 ns       3104
Analysis/symbolTable/32768                           2313859 ns    2313755 ns        301
Analysis/symbolTable/262144                         20582734 ns   20580290 ns         34
Analysis/symbolTable/2097152                       207555204 ns  207544488 ns          3
Analysis/symbolTable/10000000                     1111381830 ns 1111329506 ns          1
Analysis/symbolTable_BigO                             110.60 N      110.60 N
Analysis/symbolTable_RMS                                   6 %           6 %
AttributesBench/sameString/10                          25400 ns      25395 ns      27578
AttributesBench/sameString/64                          29777 ns      29770 ns      23165
AttributesBench/sameString/512                         67519 ns      67508 ns      10391
AttributesBench/sameString/4096                       368952 ns     368882 ns       1901
AttributesBench/sameString/32768                     2616887 ns    2616441 ns        268
AttributesBench/sameString/262144                   20386755 ns   20382261 ns         35
AttributesBench/sameString/2097152                 160800051 ns  160770866 ns          4
AttributesBench/sameString/10000000                789746984 ns  789711748 ns          1
AttributesBench/sameString_BigO                        78.88 N       78.87 N
AttributesBench/sameString_RMS                             1 %           1 %
AttributesBench/newString/10                           26939 ns      26938 ns      26077
AttributesBench/newString/64                           38844 ns      38835 ns      18031
AttributesBench/newString/512                         176931 ns     176919 ns       3973
AttributesBench/newString/4096                       1362397 ns    1362302 ns        509
AttributesBench/newString/32768                     11228015 ns   11227408 ns         62
AttributesBench/newString/262144                   111172923 ns  111167549 ns          6
AttributesBench/newString/2097152                 1591535825 ns 1591347447 ns          1
AttributesBench/newString/10000000                8887091339 ns 8886357582 ns          1
AttributesBench/newString_BigO                        882.93 N      882.86 N
AttributesBench/newString_RMS                              8 %           8 %
AttributesBench/sameStringNoThreading/10               25093 ns      25091 ns      27714
AttributesBench/sameStringNoThreading/64               29761 ns      29759 ns      23732
AttributesBench/sameStringNoThreading/512              67391 ns      67385 ns      10772
AttributesBench/sameStringNoThreading/4096            362256 ns     362239 ns       1985
AttributesBench/sameStringNoThreading/32768          2653860 ns    2653773 ns        263
AttributesBench/sameStringNoThreading/262144        20414147 ns   20412688 ns         34
AttributesBench/sameStringNoThreading/2097152      161449035 ns  161410387 ns          4
AttributesBench/sameStringNoThreading/10000000     812195579 ns  812055621 ns          1
AttributesBench/sameStringNoThreading_BigO             81.04 N       81.02 N
AttributesBench/sameStringNoThreading_RMS                  2 %           2 %
AttributesBench/newStringNoThreading/10                25899 ns      25894 ns      26880
AttributesBench/newStringNoThreading/64                33426 ns      33420 ns      20941
AttributesBench/newStringNoThreading/512              111491 ns     111464 ns       6322
AttributesBench/newStringNoThreading/4096             799421 ns     799274 ns        871
AttributesBench/newStringNoThreading/32768           6615350 ns    6615040 ns        107
AttributesBench/newStringNoThreading/262144         60571886 ns   60568929 ns         10
AttributesBench/newStringNoThreading/2097152      1011572721 ns 1011488998 ns          1
AttributesBench/newStringNoThreading/10000000     5860333524 ns 5859865437 ns          1
AttributesBench/newStringNoThreading_BigO             581.43 N      581.38 N
AttributesBench/newStringNoThreading_RMS                   9 %           9 %
AttributesBench/sameStringMultithreaded/1             177082 ns     132487 ns       5349
AttributesBench/sameStringMultithreaded/8             188428 ns     138520 ns       4991
AttributesBench/sameStringMultithreaded/64            217087 ns     158959 ns       4453
AttributesBench/sameStringMultithreaded/512           401011 ns     186276 ns       3759
AttributesBench/sameStringMultithreaded/4096         1692038 ns     197919 ns       3654
AttributesBench/sameStringMultithreaded/10000        3872841 ns     204502 ns       1000
AttributesBench/sameStringMultithreaded_BigO          391.99 N       25.31 N
AttributesBench/sameStringMultithreaded_RMS               15 %          77 %
AttributesBench/newStringMultithreaded/1              176590 ns     131450 ns       5397
AttributesBench/newStringMultithreaded/8              225039 ns     158980 ns       4537
AttributesBench/newStringMultithreaded/64             270241 ns     185683 ns       3781
AttributesBench/newStringMultithreaded/512           1787474 ns     206312 ns       3342
AttributesBench/newStringMultithreaded/4096         13267092 ns     219467 ns       1000
AttributesBench/newStringMultithreaded/10000        31691415 ns     228920 ns        100
AttributesBench/newStringMultithreaded_BigO          3179.93 N       28.25 N
```

```
AttributesBench/newStringMultithreaded_RMS                          2 %           77 %
AttributesBench/newStringEachMultithreaded/1                   181078 ns      134121 ns         5180
AttributesBench/newStringEachMultithreaded/8                   217968 ns      159165 ns         4343
AttributesBench/newStringEachMultithreaded/64                  393130 ns      197247 ns         3421
AttributesBench/newStringEachMultithreaded/512                2220815 ns      212651 ns         3259
AttributesBench/newStringEachMultithreaded/4096              15542558 ns      238015 ns         1000
AttributesBench/newStringEachMultithreaded/10000             37028639 ns      262253 ns          100
AttributesBench/newStringEachMultithreaded_BigO               3717.53 N        31.79 N
AttributesBench/newStringEachMultithreaded_RMS                      2 %           75 %
AttributesBench/setAttrRaw/1                                      352 ns         352 ns      1894864
AttributesBench/setAttrRaw/8                                     1935 ns        1934 ns       362435
AttributesBench/setAttrRaw/64                                   14422 ns       14422 ns        48335
AttributesBench/setAttrRaw/512                                 114092 ns      114079 ns         6155
AttributesBench/setAttrRaw/4096                                917976 ns      917885 ns          768
AttributesBench/setAttrRaw/32768                              7267919 ns     7267427 ns           96
AttributesBench/setAttrRaw/100000                            22257550 ns    22255369 ns           31
AttributesBench/setAttrRaw_BigO                                222.50 N       222.48 N
AttributesBench/setAttrRaw_RMS                                      0 %            0 %
AttributesBench/setAttrProp/1                                    1.27 ns        1.27 ns    552856899
AttributesBench/setAttrProp/8                                    5.57 ns        5.57 ns    125431956
AttributesBench/setAttrProp/64                                   40.4 ns        40.3 ns     17349876
AttributesBench/setAttrProp/512                                   326 ns         326 ns      2139325
AttributesBench/setAttrProp/4096                                 2589 ns        2588 ns       270117
AttributesBench/setAttrProp/32768                               20657 ns       20655 ns        33859
AttributesBench/setAttrProp/100000                              62989 ns       62984 ns        11150
AttributesBench/setAttrProp_BigO                                 0.63 N         0.63 N
AttributesBench/setAttrProp_RMS                                     0 %            0 %
AttributesBench/setProp/1                                        1.27 ns        1.27 ns    527999624
AttributesBench/setProp/8                                        6.43 ns        6.43 ns    106786484
AttributesBench/setProp/64                                       41.7 ns        41.7 ns     16801887
AttributesBench/setProp/512                                       332 ns         332 ns      2105804
AttributesBench/setProp/4096                                     2591 ns        2591 ns       270559
AttributesBench/setProp/32768                                   20664 ns       20663 ns        33868
AttributesBench/setProp/262144                                 165411 ns      165400 ns         4230
AttributesBench/setProp/1000000                                631175 ns      631128 ns         1109
AttributesBench/setProp_BigO                                     0.63 N         0.63 N
AttributesBench/setProp_RMS                                         0 %            0 %
AttributesBench/setPropHoist/1                                  0.947 ns       0.947 ns    740202993
AttributesBench/setPropHoist/8                                   5.05 ns        5.05 ns    138881195
AttributesBench/setPropHoist/64                                  40.3 ns        40.3 ns     17345436
AttributesBench/setPropHoist/512                                  325 ns         325 ns      2153624
AttributesBench/setPropHoist/4096                                2586 ns        2585 ns       270515
AttributesBench/setPropHoist/32768                              20703 ns       20702 ns        33857
AttributesBench/setPropHoist/262144                            165570 ns      165564 ns         4225
AttributesBench/setPropHoist/1000000                           631529 ns      631485 ns         1112
AttributesBench/setPropHoist_BigO                                0.63 N         0.63 N
AttributesBench/setPropHoist_RMS                                    0 %            0 %
ConstantFolding/folding/1                                        3895 ns        3832 ns       182475
ConstantFolding/folding/8                                       15109 ns       15069 ns        46046
ConstantFolding/folding/64                                     114518 ns      114466 ns         6091
ConstantFolding/folding/512                                    508223 ns      508155 ns         1390
ConstantFolding/folding/4096                                  3223717 ns     3223398 ns          214
ConstantFolding/folding/10000                                 7825265 ns     7824889 ns           90
ConstantFolding/folding_BigO                                   783.68 N       783.64 N
ConstantFolding/folding_RMS                                         3 %            3 %
ConstantFolding/opFolder/1                                       5910 ns        5866 ns       119213
ConstantFolding/opFolder/8                                      20958 ns       20932 ns        33807
ConstantFolding/opFolder/64                                    153989 ns      153958 ns         4553
ConstantFolding/opFolder/512                                   742573 ns      742498 ns          953
ConstantFolding/opFolder/4096                                 4734908 ns     4734590 ns          145
ConstantFolding/opFolder/10000                               11462842 ns    11462018 ns           63
ConstantFolding/opFolder_BigO                                 1148.40 N      1148.32 N
ConstantFolding/opFolder_RMS                                        3 %            3 %
```

```
ConstantFolding/llvm_folding/1                      1456 ns         1381 ns       507760
ConstantFolding/llvm_folding/8                      4257 ns         4180 ns       168245
ConstantFolding/llvm_folding/64                    26929 ns        26840 ns        26297
ConstantFolding/llvm_folding/512                  156498 ns       156342 ns         4432
ConstantFolding/llvm_folding/1000                 289457 ns       289321 ns         2429
ConstantFolding/llvm_folding_BigO                 293.25 N        293.07 N
ConstantFolding/llvm_folding_RMS                       5 %             5 %
Cloning/cloneOps/10                                 2220 ns         2220 ns       318436
Cloning/cloneOps/64                                14691 ns        14690 ns        46549
Cloning/cloneOps/512                              136508 ns       136502 ns         5122
Cloning/cloneOps/4096                            1242184 ns      1242098 ns          565
Cloning/cloneOps/32768                          10503352 ns     10502102 ns           66
Cloning/cloneOps/262144                        108813716 ns    108810122 ns            5
Cloning/cloneOps/2097152                      1521077252 ns   1521011265 ns            1
Cloning/cloneOps/10000000                     7660330775 ns   7659616760 ns            1
Cloning/cloneOps_BigO                             764.08 N        764.01 N
Cloning/cloneOps_RMS                                   4 %             4 %
CreateOps/simple/10                                 1536 ns         1536 ns       460046
CreateOps/simple/64                                 9646 ns         9644 ns        72357
CreateOps/simple/512                               78018 ns        78000 ns         9087
CreateOps/simple/4096                             633086 ns       633009 ns         1105
CreateOps/simple/32768                           5075687 ns      5074757 ns          141
CreateOps/simple/262144                         39982401 ns     39974803 ns           17
CreateOps/simple/2097152                       360437464 ns    360353166 ns            2
CreateOps/simple/10000000                     2086232311 ns   2086075480 ns            1
CreateOps/simple_BigO                             207.04 N        207.02 N
CreateOps/simple_RMS                                   9 %             9 %
CreateOps/hoistedOpState/10                         1428 ns         1428 ns       493202
CreateOps/hoistedOpState/64                         9086 ns         9086 ns        76665
CreateOps/hoistedOpState/512                       72140 ns        72131 ns         9599
CreateOps/hoistedOpState/4096                     578253 ns       578227 ns         1203
CreateOps/hoistedOpState/32768                   4646163 ns      4645889 ns          152
CreateOps/hoistedOpState/262144                 37132617 ns     37129104 ns           19
CreateOps/hoistedOpState/2097152               294337820 ns    294303042 ns            2
CreateOps/hoistedOpState/10000000             1415526483 ns   1415445417 ns            1
CreateOps/hoistedOpState_BigO                     141.50 N        141.49 N
CreateOps/hoistedOpState_RMS                           0 %             0 %
CreateOps/withInsert/10                             1438 ns         1438 ns       457959
CreateOps/withInsert/64                             7362 ns         7362 ns       109665
CreateOps/withInsert/512                           48655 ns        48653 ns        13639
CreateOps/withInsert/4096                         399650 ns       399628 ns         1781
CreateOps/withInsert/32768                       3183901 ns      3183579 ns          220
CreateOps/withInsert/262144                     25701754 ns     25699965 ns           28
CreateOps/withInsert/2097152                   203392387 ns    203380240 ns            3
CreateOps/withInsert/10000000                 1091894863 ns   1091801688 ns            1
CreateOps/withInsert_BigO                         108.67 N        108.66 N
CreateOps/withInsert_RMS                               5 %             5 %
CreateOps/simpleRegistered/10                       1071 ns         1070 ns       637199
CreateOps/simpleRegistered/64                       7944 ns         7943 ns       100889
CreateOps/simpleRegistered/512                     75907 ns        75901 ns         9140
CreateOps/simpleRegistered/4096                   602203 ns       602160 ns         1171
CreateOps/simpleRegistered/32768                 5002004 ns      5001232 ns          100
CreateOps/simpleRegistered/262144               38527344 ns     38525541 ns           18
CreateOps/simpleRegistered/2097152             314170408 ns    314147116 ns            2
CreateOps/simpleRegistered/10000000           1486016787 ns   1485892910 ns            1
CreateOps/simpleRegistered_BigO                   148.65 N        148.64 N
CreateOps/simpleRegistered_RMS                         0 %             0 %
CreateOps/withInsertRegistered/10                   1529 ns         1529 ns       446050
CreateOps/withInsertRegistered/64                   8186 ns         8186 ns        92710
CreateOps/withInsertRegistered/512                 60538 ns        60532 ns        11322
CreateOps/withInsertRegistered/4096               469575 ns       469520 ns         1447
CreateOps/withInsertRegistered/32768             3768329 ns      3768200 ns          183
CreateOps/withInsertRegistered/262144           30147703 ns     30146259 ns           23
```

```
CreateOps/withInsertRegistered/2097152                     240048733 ns    240016824 ns            3
CreateOps/withInsertRegistered/10000000                   1287778683 ns   1287710159 ns            1
CreateOps/withInsertRegistered_BigO                           128.17 N        128.16 N
CreateOps/withInsertRegistered_RMS                                 5 %             5 %
CreateOps/llvm_withInsertRegistered/10                           621 ns          621 ns      1310205
CreateOps/llvm_withInsertRegistered/64                          3399 ns         3399 ns       208887
CreateOps/llvm_withInsertRegistered/512                        26754 ns        26751 ns        26019
CreateOps/llvm_withInsertRegistered/4096                      214032 ns       214024 ns         3267
CreateOps/llvm_withInsertRegistered/32768                    1740016 ns      1739836 ns          409
CreateOps/llvm_withInsertRegistered/262144                  13703840 ns     13703383 ns           51
CreateOps/llvm_withInsertRegistered/2097152                116423890 ns    116413778 ns            7
CreateOps/llvm_withInsertRegistered/10000000               508718534 ns    508690844 ns            1
CreateOps/llvm_withInsertRegistered_BigO                       51.07 N         51.07 N
CreateOps/llvm_withInsertRegistered_RMS                            4 %             4 %
CreateOps/simpleConstant/10                                     4259 ns         4259 ns       166102
CreateOps/simpleConstant/64                                    27216 ns        27212 ns        25925
CreateOps/simpleConstant/512                                  215266 ns       215255 ns         3259
CreateOps/simpleConstant/4096                                1706406 ns      1706316 ns          407
CreateOps/simpleConstant/32768                              13536332 ns     13535658 ns           51
CreateOps/simpleConstant/262144                            109289148 ns    109280023 ns            6
CreateOps/simpleConstant/2097152                           872710808 ns    872649757 ns            1
CreateOps/simpleConstant/10000000                         4174540562 ns   4174213028 ns            1
CreateOps/simpleConstant_BigO                                 417.40 N        417.37 N
CreateOps/simpleConstant_RMS                                       0 %             0 %
CreateOps/simpleRegisteredConstant/10                           3584 ns         3583 ns       195947
CreateOps/simpleRegisteredConstant/64                          22858 ns        22857 ns        30869
CreateOps/simpleRegisteredConstant/512                        179839 ns       179822 ns         3900
CreateOps/simpleRegisteredConstant/4096                      1436457 ns      1436281 ns          495
CreateOps/simpleRegisteredConstant/32768                    11687039 ns     11686423 ns           61
CreateOps/simpleRegisteredConstant/262144                   93986445 ns     93981382 ns            8
CreateOps/simpleRegisteredConstant/2097152                 734716294 ns    734631856 ns            1
CreateOps/simpleRegisteredConstant/10000000               3516873944 ns   3516440595 ns            1
CreateOps/simpleRegisteredConstant_BigO                       351.64 N        351.59 N
CreateOps/simpleRegisteredConstant_RMS                             0 %             0 %
DialectConversion/noPatterns/1                                  2114 ns         2058 ns       341049
DialectConversion/noPatterns/8                                  3206 ns         3144 ns       223480
DialectConversion/noPatterns/64                                10719 ns        10667 ns        67748
DialectConversion/noPatterns/512                               67336 ns        67225 ns        10457
DialectConversion/noPatterns/4096                             522999 ns       522861 ns         1337
DialectConversion/noPatterns/32768                           4157336 ns      4156524 ns          168
DialectConversion/noPatterns/262144                         34530702 ns     34527365 ns           20
DialectConversion/noPatterns/1000000                       141217638 ns    141213232 ns            5
DialectConversion/noPatterns_BigO                             140.59 N        140.59 N
DialectConversion/noPatterns_RMS                                   4 %             4 %
DialectConversion/toLLVM/1                                     26222 ns        26240 ns        26347
DialectConversion/toLLVM/8                                     52543 ns        52546 ns        13238
DialectConversion/toLLVM/64                                   270734 ns       270727 ns         2555
DialectConversion/toLLVM/512                                 1942653 ns      1942429 ns          355
DialectConversion/toLLVM/4096                               15547873 ns     15547373 ns           45
DialectConversion/toLLVM/32768                             135306178 ns    135301871 ns            5
DialectConversion/toLLVM/262144                           1349905322 ns   1349700436 ns            1
DialectConversion/toLLVM/1000000                          6221941548 ns   6221300959 ns            1
DialectConversion/toLLVM_BigO                                6150.91 N       6150.26 N
DialectConversion/toLLVM_RMS                                      10 %            10 %
GreedyRewriter/empty/1                                           160 ns          160 ns      4317770
GreedyRewriter/empty/8                                           310 ns          310 ns      2242470
GreedyRewriter/empty/64                                         1999 ns         1998 ns       344037
GreedyRewriter/empty/512                                       18588 ns        18584 ns        37692
GreedyRewriter/empty/4096                                     196155 ns       196116 ns         3571
GreedyRewriter/empty/32768                                   1562059 ns      1561765 ns          453
GreedyRewriter/empty/262144                                 14990618 ns     14987106 ns           47
GreedyRewriter/empty/2097152                               161013784 ns    160978620 ns            4
GreedyRewriter/empty/10000000                             1160105739 ns   1159997571 ns            1
```

GreedyRewriter/empty_BigO	114.32	N	114.31	N	
GreedyRewriter/empty_RMS	18	%	18	%	
GreedyRewriter/withCanonicalizationPatterns/10	40997	ns	40995	ns	16619
GreedyRewriter/withCanonicalizationPatterns/64	43101	ns	43100	ns	16070
GreedyRewriter/withCanonicalizationPatterns/512	61193	ns	61189	ns	11507
GreedyRewriter/withCanonicalizationPatterns/4096	229037	ns	229017	ns	2960
GreedyRewriter/withCanonicalizationPatterns/32768	1687291	ns	1686925	ns	430
GreedyRewriter/withCanonicalizationPatterns/262144	15046323	ns	15042271	ns	47
GreedyRewriter/withCanonicalizationPatterns/2097152	166651302	ns	166609746	ns	4
GreedyRewriter/withCanonicalizationPatterns/10000000	1205316641	ns	1205235646	ns	1
GreedyRewriter/withCanonicalizationPatterns_BigO	118.76	N	118.75	N	
GreedyRewriter/withCanonicalizationPatterns_RMS	17	%	17	%	
GreedyRewriter/withPatterns/100	48255	ns	48244	ns	14615
GreedyRewriter/withPatterns/512	86790	ns	86771	ns	7815
GreedyRewriter/withPatterns/4096	630096	ns	630079	ns	1172
GreedyRewriter/withPatterns/10000	1454875	ns	1454797	ns	480
GreedyRewriter/withPatterns_BigO	146.77	N	146.76	N	
GreedyRewriter/withPatterns_RMS	4	%	4	%	
InterfaceBench/vectorTraveralCallInterfaceMethod/10	99.8	ns	99.8	ns	7047449
InterfaceBench/vectorTraveralCallInterfaceMethod/64	591	ns	591	ns	1183230
InterfaceBench/vectorTraveralCallInterfaceMethod/512	4638	ns	4638	ns	150472
InterfaceBench/vectorTraveralCallInterfaceMethod/4096	37237	ns	37229	ns	18789
InterfaceBench/vectorTraveralCallInterfaceMethod/32768	300670	ns	300651	ns	2333
InterfaceBench/vectorTraveralCallInterfaceMethod/262144	2643092	ns	2642860	ns	263
InterfaceBench/vectorTraveralCallInterfaceMethod/2097152	20712952	ns	20708857	ns	34
InterfaceBench/vectorTraveralCallInterfaceMethod/10000000	98550537	ns	98542763	ns	7
InterfaceBench/vectorTraveralCallInterfaceMethod_BigO	9.86	N	9.86	N	
InterfaceBench/vectorTraveralCallInterfaceMethod_RMS	0	%	0	%	
InterfaceBench/llvm_vectorTraveralCallInterfaceMethod/10	15.0	ns	15.0	ns	46612990
InterfaceBench/llvm_vectorTraveralCallInterfaceMethod/64	93.3	ns	93.3	ns	7470909
InterfaceBench/llvm_vectorTraveralCallInterfaceMethod/512	699	ns	699	ns	995669
InterfaceBench/llvm_vectorTraveralCallInterfaceMethod/4096	5522	ns	5521	ns	126964
InterfaceBench/llvm_vectorTraveralCallInterfaceMethod/10000	15494	ns	15493	ns	45167
InterfaceBench/llvm_vectorTraveralCallInterfaceMethod_BigO	1.52	N	1.52	N	
InterfaceBench/llvm_vectorTraveralCallInterfaceMethod_RMS	8	%	8	%	
LoopUnrolling/unroll/1	4502	ns	4410	ns	158967
LoopUnrolling/unroll/8	23767	ns	23618	ns	29563
LoopUnrolling/unroll/64	172746	ns	172507	ns	4052
LoopUnrolling/unroll/512	1322454	ns	1321995	ns	534
LoopUnrolling/unroll/1000	2545860	ns	2545325	ns	276
LoopUnrolling/unroll_BigO	2554.05	N	2553.43	N	
LoopUnrolling/unroll_RMS	1	%	1	%	
LoopUnrolling/llvm_unroll/1	36144	ns	36123	ns	19441
LoopUnrolling/llvm_unroll/8	255243	ns	255368	ns	2735
LoopUnrolling/llvm_unroll/64	2700037	ns	2700070	ns	259
LoopUnrolling/llvm_unroll/512	73407469	ns	73402333	ns	10
LoopUnrolling/llvm_unroll/1000	260718746	ns	260696177	ns	3
LoopUnrolling/llvm_unroll_BigO	235708.03	N	235688.13	N	
LoopUnrolling/llvm_unroll_RMS	36	%	36	%	
ParserPrinter/parseTextualIR/10	61549	ns	61546	ns	11338
ParserPrinter/parseTextualIR/64	179556	ns	179544	ns	3894
ParserPrinter/parseTextualIR/512	1220245	ns	1220158	ns	573
ParserPrinter/parseTextualIR/4096	9856960	ns	9855832	ns	71
ParserPrinter/parseTextualIR/32768	82664124	ns	82658142	ns	8
ParserPrinter/parseTextualIR/262144	995924502	ns	995851929	ns	1
ParserPrinter/parseTextualIR/1000000	4324001717	ns	4323683749	ns	1
ParserPrinter/parseTextualIR_BigO	4288.45	N	4288.13	N	
ParserPrinter/parseTextualIR_RMS	7	%	7	%	
ParserPrinter/parseBytecodeIR/10	47988	ns	47984	ns	14588
ParserPrinter/parseBytecodeIR/64	100307	ns	100281	ns	6952
ParserPrinter/parseBytecodeIR/512	573133	ns	573025	ns	1218
ParserPrinter/parseBytecodeIR/4096	4586582	ns	4585625	ns	153
ParserPrinter/parseBytecodeIR/32768	37373868	ns	37371286	ns	19

ParserPrinter/parseBytecodeIR/262144	420982484	ns	420959945	ns	2
ParserPrinter/parseBytecodeIR/1000000	2034472944	ns	2034323491	ns	1
ParserPrinter/parseBytecodeIR_BigO	2006.03	N	2005.89	N	
ParserPrinter/parseBytecodeIR_RMS	12	%	12	%	
ParserPrinter/printTextualIR/10	14484	ns	14482	ns	48128
ParserPrinter/printTextualIR/64	62662	ns	62660	ns	11078
ParserPrinter/printTextualIR/512	463614	ns	463600	ns	1507
ParserPrinter/printTextualIR/4096	3802635	ns	3802384	ns	184
ParserPrinter/printTextualIR/32768	31159720	ns	31152274	ns	22
ParserPrinter/printTextualIR/262144	298570499	ns	298498153	ns	2
ParserPrinter/printTextualIR/1000000	1246963499	ns	1246833554	ns	1
ParserPrinter/printTextualIR_BigO	1239.72	N	1239.58	N	
ParserPrinter/printTextualIR_RMS	5	%	5	%	
ParserPrinter/printBytecodeIR/10	12809	ns	12805	ns	53762
ParserPrinter/printBytecodeIR/64	49801	ns	49790	ns	13943
ParserPrinter/printBytecodeIR/512	408265	ns	408185	ns	1735
ParserPrinter/printBytecodeIR/4096	3312854	ns	3312253	ns	209
ParserPrinter/printBytecodeIR/32768	28967742	ns	28958923	ns	23
ParserPrinter/printBytecodeIR/262144	299572774	ns	299555963	ns	2
ParserPrinter/printBytecodeIR/1000000	1386730152	ns	1386637183	ns	1
ParserPrinter/printBytecodeIR_BigO	1370.55	N	1370.45	N	
ParserPrinter/printBytecodeIR_RMS	10	%	10	%	
IRWalk/blockTraveral/10	17.5	ns	17.5	ns	39881680
IRWalk/blockTraveral/64	110	ns	110	ns	6379697
IRWalk/blockTraveral/512	1596	ns	1595	ns	439735
IRWalk/blockTraveral/4096	10195	ns	10194	ns	68860
IRWalk/blockTraveral/32768	81073	ns	81055	ns	8650
IRWalk/blockTraveral/262144	1841230	ns	1840690	ns	381
IRWalk/blockTraveral/2097152	16323138	ns	16319776	ns	43
IRWalk/blockTraveral/10000000	80215734	ns	80210678	ns	9
IRWalk/blockTraveral_BigO	8.01	N	8.01	N	
IRWalk/blockTraveral_RMS	2	%	2	%	
IRWalk/vectorTraveral/10	4.06	ns	4.06	ns	171981038
IRWalk/vectorTraveral/64	26.4	ns	26.4	ns	26457066
IRWalk/vectorTraveral/512	232	ns	232	ns	3020721
IRWalk/vectorTraveral/4096	1849	ns	1849	ns	379201
IRWalk/vectorTraveral/32768	14762	ns	14759	ns	47414
IRWalk/vectorTraveral/262144	114403	ns	114397	ns	6116
IRWalk/vectorTraveral/2097152	1028901	ns	1028851	ns	685
IRWalk/vectorTraveral/10000000	6133232	ns	6132622	ns	114
IRWalk/vectorTraveral_BigO	0.61	N	0.61	N	
IRWalk/vectorTraveral_RMS	10	%	10	%	
IRWalk/simpleWalk/10	62.6	ns	62.6	ns	11116850
IRWalk/simpleWalk/64	350	ns	350	ns	1999507
IRWalk/simpleWalk/512	2729	ns	2729	ns	256518
IRWalk/simpleWalk/4096	21833	ns	21831	ns	32038
IRWalk/simpleWalk/32768	176236	ns	176223	ns	3985
IRWalk/simpleWalk/262144	1742919	ns	1742834	ns	404
IRWalk/simpleWalk/2097152	14132705	ns	14131555	ns	49
IRWalk/simpleWalk/10000000	67449778	ns	67445037	ns	10
IRWalk/simpleWalk_BigO	6.74	N	6.74	N	
IRWalk/simpleWalk_RMS	0	%	0	%	
IRWalk/filteredOps/10	17.1	ns	17.1	ns	40873782
IRWalk/filteredOps/64	113	ns	113	ns	6174719
IRWalk/filteredOps/512	1603	ns	1603	ns	431125
IRWalk/filteredOps/4096	9621	ns	9620	ns	72549
IRWalk/filteredOps/32768	76871	ns	76866	ns	9082
IRWalk/filteredOps/262144	1558707	ns	1558516	ns	452
IRWalk/filteredOps/2097152	13222558	ns	13221336	ns	53
IRWalk/filteredOps/10000000	62731864	ns	62725928	ns	11
IRWalk/filteredOps_BigO	6.27	N	6.27	N	
IRWalk/filteredOps_RMS	1	%	1	%	
IRWalk/nestedRegion/10	84.7	ns	84.7	ns	8264952

```
IRWalk/nestedRegion/16                                         125 ns          125 ns      5587643
IRWalk/nestedRegion/64                                         454 ns          454 ns      1531392
IRWalk/nestedRegion/256                                       1828 ns         1828 ns       384758
IRWalk/nestedRegion/1024                                      7129 ns         7129 ns        91981
IRWalk/nestedRegion/4096                                     30161 ns        30159 ns        23193
IRWalk/nestedRegion/8000                                     57663 ns        57658 ns        12012
IRWalk/nestedRegion_BigO                                      7.24 N          7.24 N
IRWalk/nestedRegion_RMS                                          2 %             2 %
IRWalk/llvm_blockTraversal/10                                 7.48 ns         7.48 ns     93670794
IRWalk/llvm_blockTraversal/64                                 69.4 ns         69.4 ns     10072389
IRWalk/llvm_blockTraversal/512                                1476 ns         1476 ns       469258
IRWalk/llvm_blockTraversal/4096                               9975 ns         9975 ns        70587
IRWalk/llvm_blockTraversal/32768                             80465 ns        80459 ns         8673
IRWalk/llvm_blockTraversal/262144                          1803648 ns      1803592 ns          390
IRWalk/llvm_blockTraversal/2097152                        16225756 ns     16225254 ns           43
IRWalk/llvm_blockTraversal/10000000                       78978276 ns     78975659 ns            9
IRWalk/llvm_blockTraversal_BigO                               7.89 N          7.89 N
IRWalk/llvm_blockTraversal_RMS                                   1 %             1 %
IRWalk/vectorTraveralOpCastFail/10                            7.36 ns         7.36 ns     95088104
IRWalk/vectorTraveralOpCastFail/64                            45.7 ns         45.7 ns     15317168
IRWalk/vectorTraveralOpCastFail/512                            476 ns          476 ns      1450136
IRWalk/vectorTraveralOpCastFail/4096                          3790 ns         3790 ns       184584
IRWalk/vectorTraveralOpCastFail/32768                        39812 ns        39809 ns        17634
IRWalk/vectorTraveralOpCastFail/262144                      975434 ns       975405 ns          720
IRWalk/vectorTraveralOpCastFail/2097152                    8580938 ns      8580456 ns           81
IRWalk/vectorTraveralOpCastFail/10000000                  40768999 ns     40764746 ns           17
IRWalk/vectorTraveralOpCastFail_BigO                          4.08 N          4.08 N
IRWalk/vectorTraveralOpCastFail_RMS                              1 %             1 %
IRWalk/vectorTraveralOpCastSuccess/10                         7.39 ns         7.39 ns     93692158
IRWalk/vectorTraveralOpCastSuccess/64                         45.7 ns         45.7 ns     15323919
IRWalk/vectorTraveralOpCastSuccess/512                         523 ns          523 ns      1294067
IRWalk/vectorTraveralOpCastSuccess/4096                       3785 ns         3785 ns       186404
IRWalk/vectorTraveralOpCastSuccess/32768                     39972 ns        39970 ns        17502
IRWalk/vectorTraveralOpCastSuccess/262144                   977336 ns       977306 ns          720
IRWalk/vectorTraveralOpCastSuccess/2097152                 8450122 ns      8447973 ns           81
IRWalk/vectorTraveralOpCastSuccess/10000000               40680953 ns     40678330 ns           17
IRWalk/vectorTraveralOpCastSuccess_BigO                       4.07 N          4.07 N
IRWalk/vectorTraveralOpCastSuccess_RMS                           1 %             1 %
IRWalk/vectorTraveralOpIsaSuccess/10                          7.02 ns         7.02 ns     99761463
IRWalk/vectorTraveralOpIsaSuccess/64                          47.6 ns         47.6 ns     14778467
IRWalk/vectorTraveralOpIsaSuccess/512                          420 ns          420 ns      1628629
IRWalk/vectorTraveralOpIsaSuccess/4096                        4313 ns         4313 ns       161931
IRWalk/vectorTraveralOpIsaSuccess/32768                      38468 ns        38465 ns        17969
IRWalk/vectorTraveralOpIsaSuccess/262144                    983971 ns       983943 ns          708
IRWalk/vectorTraveralOpIsaSuccess/2097152                  8727375 ns      8726866 ns           79
IRWalk/vectorTraveralOpIsaSuccess/10000000                41350491 ns     41343429 ns           17
IRWalk/vectorTraveralOpIsaSuccess_BigO                        4.14 N          4.14 N
IRWalk/vectorTraveralOpIsaSuccess_RMS                            1 %             1 %
IRWalk/vectorTraveralWithoutInterfaceCastFail/10              39.2 ns         39.2 ns     17857367
IRWalk/vectorTraveralWithoutInterfaceCastFail/64               254 ns          254 ns      2761824
IRWalk/vectorTraveralWithoutInterfaceCastFail/512             2132 ns         2131 ns       328156
IRWalk/vectorTraveralWithoutInterfaceCastFail/4096           16273 ns        16272 ns        42906
IRWalk/vectorTraveralWithoutInterfaceCastFail/32768         131320 ns       131283 ns         5341
IRWalk/vectorTraveralWithoutInterfaceCastFail/262144       1665353 ns      1664450 ns          422
IRWalk/vectorTraveralWithoutInterfaceCastFail/2097152     13372446 ns     13369410 ns           52
IRWalk/vectorTraveralWithoutInterfaceCastFail/10000000    63311417 ns     63307361 ns           11
IRWalk/vectorTraveralWithoutInterfaceCastFail_BigO            6.33 N          6.33 N
IRWalk/vectorTraveralWithoutInterfaceCastFail_RMS                0 %             0 %
IRWalk/vectorTraveralWithInterfaceCastFail/10                 47.3 ns         47.3 ns     14787937
IRWalk/vectorTraveralWithInterfaceCastFail/64                  308 ns          308 ns      2278677
IRWalk/vectorTraveralWithInterfaceCastFail/512                2458 ns         2458 ns       285080
IRWalk/vectorTraveralWithInterfaceCastFail/4096              19516 ns        19515 ns        35921
IRWalk/vectorTraveralWithInterfaceCastFail/32768            156901 ns       156861 ns         4470
```

```
IRWalk/vectorTraveralWithInterfaceCastFail/262144          2045579 ns     2045320 ns          338
IRWalk/vectorTraveralWithInterfaceCastFail/2097152        16461268 ns    16455022 ns           43
IRWalk/vectorTraveralWithInterfaceCastFail/10000000       78230340 ns    78221750 ns            9
IRWalk/vectorTraveralWithInterfaceCastFail_BigO               7.82 N         7.82 N
IRWalk/vectorTraveralWithInterfaceCastFail_RMS                   0 %            0 %
IRWalk/vectorTraveralWithInterfaceCastSuccess/10              85.7 ns        85.7 ns      8214303
IRWalk/vectorTraveralWithInterfaceCastSuccess/64               489 ns         489 ns      1431791
IRWalk/vectorTraveralWithInterfaceCastSuccess/512             3933 ns        3933 ns       179271
IRWalk/vectorTraveralWithInterfaceCastSuccess/4096           30951 ns       30949 ns        22630
IRWalk/vectorTraveralWithInterfaceCastSuccess/32768         248877 ns      248870 ns         2805
IRWalk/vectorTraveralWithInterfaceCastSuccess/262144       2338713 ns     2337678 ns          300
IRWalk/vectorTraveralWithInterfaceCastSuccess/2097152     18149006 ns    18146236 ns           38
IRWalk/vectorTraveralWithInterfaceCastSuccess/10000000    88021965 ns    88009654 ns            8
IRWalk/vectorTraveralWithInterfaceCastSuccess_BigO            8.80 N         8.79 N
IRWalk/vectorTraveralWithInterfaceCastSuccess_RMS                1 %            1 %
IRWalk/vectorTraveralOpTraitFail/10                           83.6 ns        83.6 ns      8404678
IRWalk/vectorTraveralOpTraitFail/64                            543 ns         543 ns      1292619
IRWalk/vectorTraveralOpTraitFail/512                          4284 ns        4284 ns       163011
IRWalk/vectorTraveralOpTraitFail/4096                        34317 ns       34315 ns        20364
IRWalk/vectorTraveralOpTraitFail/32768                      278203 ns      278124 ns         2511
IRWalk/vectorTraveralOpTraitFail/262144                    2520282 ns     2519074 ns          281
IRWalk/vectorTraveralOpTraitFail/2097152                  19620514 ns    19610789 ns           36
IRWalk/vectorTraveralOpTraitFail/10000000                 93801856 ns    93792653 ns            7
IRWalk/vectorTraveralOpTraitFail_BigO                         9.38 N         9.38 N
IRWalk/vectorTraveralOpTraitFail_RMS                             0 %            0 %
IRWalk/vectorTraveralOpTraitSuccess/10                         117 ns         117 ns      5977891
IRWalk/vectorTraveralOpTraitSuccess/64                         749 ns         749 ns       935541
IRWalk/vectorTraveralOpTraitSuccess/512                       5927 ns        5927 ns       118265
IRWalk/vectorTraveralOpTraitSuccess/4096                     47596 ns       47592 ns        14704
IRWalk/vectorTraveralOpTraitSuccess/32768                   383662 ns      383566 ns         1826
IRWalk/vectorTraveralOpTraitSuccess/262144                 3441962 ns     3440234 ns          202
IRWalk/vectorTraveralOpTraitSuccess/2097152               27770850 ns    27769434 ns           25
IRWalk/vectorTraveralOpTraitSuccess/10000000             133148747 ns   133141848 ns            5
IRWalk/vectorTraveralOpTraitSuccess_BigO                     13.31 N        13.31 N
IRWalk/vectorTraveralOpTraitSuccess_RMS                          0 %            0 %
SimpleConstantFolding/folding/1                              10112 ns       10070 ns        70189
SimpleConstantFolding/folding/8                              81970 ns       81663 ns         8557
SimpleConstantFolding/folding/64                            640847 ns      638256 ns         1074
SimpleConstantFolding/folding/512                          5222675 ns     5202763 ns          135
SimpleConstantFolding/folding/4096                        42047737 ns    41843492 ns           17
SimpleConstantFolding/folding/10000                      100531426 ns   100120405 ns            7
SimpleConstantFolding/folding_BigO                        10083.92 N     10041.57 N
SimpleConstantFolding/folding_RMS                                1 %            1 %
Dynamism/noCall/10                                            6.14 ns        6.14 ns    113553610
Dynamism/noCall/64                                            39.4 ns        39.4 ns     17745295
Dynamism/noCall/512                                            325 ns         325 ns      2157076
Dynamism/noCall/4096                                          2532 ns        2532 ns       277616
Dynamism/noCall/32768                                        20200 ns       20199 ns        34687
Dynamism/noCall/262144                                      161503 ns      161477 ns         4331
Dynamism/noCall/2097152                                    1321702 ns     1321604 ns          541
Dynamism/noCall/10000000                                   6161264 ns     6160970 ns          113
Dynamism/noCall_BigO                                          0.62 N         0.62 N
Dynamism/noCall_RMS                                              1 %            1 %
Dynamism/regularCall/10                                       7.38 ns        7.38 ns     94848237
Dynamism/regularCall/64                                       40.7 ns        40.7 ns     17091447
Dynamism/regularCall/512                                       333 ns         333 ns      2155774
Dynamism/regularCall/4096                                     2537 ns        2537 ns       275792
Dynamism/regularCall/32768                                   20152 ns       20151 ns        34725
Dynamism/regularCall/262144                                 161299 ns      161295 ns         4337
Dynamism/regularCall/2097152                               1290940 ns     1290854 ns          541
Dynamism/regularCall/10000000                              6153182 ns     6152711 ns          114
Dynamism/regularCall_BigO                                     0.62 N         0.62 N
Dynamism/regularCall_RMS                                         0 %            0 %
```

```
Dynamism/regularCallNoinline/10                              16.9 ns         16.9 ns     41368407
Dynamism/regularCallNoinline/64                               112 ns          112 ns      6255407
Dynamism/regularCallNoinline/512                              799 ns          799 ns       873954
Dynamism/regularCallNoinline/4096                            6299 ns         6298 ns       110858
Dynamism/regularCallNoinline/32768                          50360 ns        50347 ns        13933
Dynamism/regularCallNoinline/262144                        402420 ns       402348 ns         1741
Dynamism/regularCallNoinline/2097152                      3215942 ns      3215367 ns          217
Dynamism/regularCallNoinline/10000000                    15373396 ns     15372190 ns           46
Dynamism/regularCallNoinline_BigO                            1.54 N          1.54 N
Dynamism/regularCallNoinline_RMS                                0 %             0 %
Dynamism/regularPointerCall/10                               6.81 ns         6.80 ns    102257870
Dynamism/regularPointerCall/64                               40.7 ns         40.7 ns     16965482
Dynamism/regularPointerCall/512                               324 ns          324 ns      2163497
Dynamism/regularPointerCall/4096                             2530 ns         2530 ns       276256
Dynamism/regularPointerCall/32768                           20109 ns        20107 ns        34720
Dynamism/regularPointerCall/262144                         160961 ns       160944 ns         4351
Dynamism/regularPointerCall/2097152                       1288637 ns      1288545 ns          543
Dynamism/regularPointerCall/10000000                      6164348 ns      6164179 ns          114
Dynamism/regularPointerCall_BigO                             0.62 N          0.62 N
Dynamism/regularPointerCall_RMS                                 0 %             0 %
Dynamism/runtimePolymorphicPointerCall/10                    19.2 ns         19.2 ns     36665574
Dynamism/runtimePolymorphicPointerCall/64                     129 ns          129 ns      5411307
Dynamism/runtimePolymorphicPointerCall/512                    958 ns          958 ns       730546
Dynamism/runtimePolymorphicPointerCall/4096                  7582 ns         7581 ns        92110
Dynamism/runtimePolymorphicPointerCall/32768                60403 ns        60399 ns        11566
Dynamism/runtimePolymorphicPointerCall/262144              483774 ns       483742 ns         1451
Dynamism/runtimePolymorphicPointerCall/2097152            3859047 ns      3858720 ns          181
Dynamism/runtimePolymorphicPointerCall/10000000          18391560 ns     18390115 ns           38
Dynamism/runtimePolymorphicPointerCall_BigO                  1.84 N          1.84 N
Dynamism/runtimePolymorphicPointerCall_RMS                      0 %             0 %
```

Listing C.2: Results for the "How Slow is MLIR?" micro-benchmarks.

Appendix D

xDSL benchmark results

D.1 Pipeline phase micro-benchmark results

D.2 Micro-benchmark results

The following section describes the procedure and provides the raw results from the xDSL micro-benchmarks derived from “How Slow is MLIR’s”.

```
git clone https://github.com/xdslproject/xdsl
make venv
python3 benchmarks/microbenchmarks all timeit
```

Listing D.1: Bash commands to download, setup the environment for, and run the benchmarks for xDSL derived from from “How Slow is MLIR”.

```
Test IRTraversal.iterate_ops ran in: 0.000326 ± 5.92e-06s // ÷ 32768
Test IRTraversal.iterate_block_ops ran in: 0.00675 ± 4.5e-05s // ÷ 32768
Test IRTraversal.walk_block_ops ran in: 0.0172 ± 9.31e-05s // ÷ 32768
Test Extensibility.interface_check_trait ran in: 3.87e-08 ± 2.79e-07s
Test Extensibility.interface_cast_trait ran in: 1.02e-07 ± 4.67e-07s
Test Extensibility.interface_check ran in: 3.71e-08 ± 2.72e-07s
Test Extensibility.trait_check ran in: 1.92e-06 ± 6.95e-07s
Test Extensibility.trait_check_optimised ran in: 2.39e-07 ± 3.51e-08s
Test Extensibility.trait_check_single ran in: 1.73e-06 ± 7.03e-07s
Test Extensibility.trait_check_neg ran in: 2.83e-06 ± 7.76e-07s
Test OpCreation.operation_create ran in: 3.77e-06 ± 9.16e-07s
Test OpCreation.operation_build ran in: 1.27e-05 ± 1.81e-06s
Test OpCreation.operation_create_optimised ran in: 3.78e-07 ± 4.42e-08s
Test OpCreation.operation_clone ran in: 0.000962 ± 1.15e-05s
Test OpCreation.operation_clone_single ran in: 6.96e-06 ± 1.55e-06s
Test OpCreation.operation_constant_init ran in: 3.34e-05 ± 3.68e-06s
Test OpCreation.operation_constant_create ran in: 1.83e-06 ± 7.74e-07s
Test ConstantFoldingSimple.20 ran in: 0.00114 ± 3.79e-05s
Test ConstantFoldingSimpleInlined.20 ran in: 0.000135 ± 2.68e-05s
```

Listing D.2: Results for the xDSL micro-benchmarks derived from “How Slow is MLIR?”, for CPython version 3.10.17.

```
Test IRTraversal.iterate_ops ran in: 0.00036 ± 8.27e-06s
Test IRTraversal.iterate_block_ops ran in: 0.00401 ± 2.31e-05s
Test IRTraversal.walk_block_ops ran in: 0.0141 ± 0.000126s
Test Extensibility.interface_check_trait ran in: 2.8e-08 ± 3e-07s
Test Extensibility.interface_check ran in: 3.02e-08 ± 2.6e-07s
Test Extensibility.trait_check ran in: 1.71e-06 ± 7.53e-07s
Test Extensibility.trait_check_optimised ran in: 1.91e-07 ± 3.34e-08s
Test Extensibility.trait_check_single ran in: 1.48e-06 ± 7.36e-07s
Test Extensibility.trait_check_neg ran in: 2.53e-06 ± 9.22e-07s
Test OpCreation.operation_create ran in: 2.84e-06 ± 8.83e-07s
Test OpCreation.operation_build ran in: 1.06e-05 ± 1.94e-06s
Test OpCreation.operation_create_optimised ran in: 2.65e-07 ± 3.72e-08s
Test OpCreation.operation_clone ran in: 0.00074 ± 1.09e-05s
Test OpCreation.operation_clone_single ran in: 5.27e-06 ± 1.1e-06s
Test OpCreation.operation_constant_init ran in: 2.77e-05 ± 3.64e-06s
Test OpCreation.operation_constant_create ran in: 1.7e-06 ± 1.24e-06s
Test ConstantFoldingSimple.20 ran in: 0.00086 ± 2.68e-05s
Test ConstantFoldingSimpleInlined.20 ran in: 9.24e-05 ± 2.15e-05s
```

Listing D.3: Results for the xDSL micro-benchmarks derived from "How Slow is MLIR?", for CPython version 3.11.12.

```
Test IRTraversal.iterate_ops ran in: 0.000318 ± 6.61e-06s
Test IRTraversal.iterate_block_ops ran in: 0.00487 ± 2.48e-05s
Test IRTraversal.walk_block_ops ran in: 0.0129 ± 7.65e-05s
Test Extensibility.interface_check_trait ran in: 3.94e-08 ± 3.1e-07s
Test Extensibility.interface_cast_trait ran in: 8.47e-08 ± 4.25e-07s
Test Extensibility.interface_check ran in: 3.97e-08 ± 2.3e-07s
Test Extensibility.trait_check ran in: 1.19e-06 ± 5.53e-07s
Test Extensibility.trait_check_optimised ran in: 2.21e-07 ± 3.09e-08s
Test Extensibility.trait_check_single ran in: 1e-06 ± 5.2e-07s
Test Extensibility.trait_check_neg ran in: 2.09e-06 ± 6.81e-07s
Test OpCreation.operation_create ran in: 2.99e-06 ± 8.08e-07s
Test OpCreation.operation_build ran in: 1.17e-05 ± 1.59e-06s
Test OpCreation.operation_create_optimised ran in: 3.65e-07 ± 3.25e-08s
Test OpCreation.operation_clone ran in: 0.000746 ± 1.11e-05s
Test OpCreation.operation_clone_single ran in: 5.2e-06 ± 1.56e-06s
Test OpCreation.operation_constant_init ran in: 2.85e-05 ± 3.54e-06s
Test OpCreation.operation_constant_create ran in: 1.91e-06 ± 1.25e-06s
Test ConstantFoldingSimple.20 ran in: 0.000892 ± 2.66e-05s
Test ConstantFoldingSimpleInlined.20 ran in: 0.000108 ± 2.51e-05s
```

Listing D.4: Results for the xDSL micro-benchmarks derived from "How Slow is MLIR?", for CPython version 3.13.3.

```
Test IRTraversal.iterate_ops ran in: 0.000152 ± 9.27e-06s
Test IRTraversal.iterate_block_ops ran in: 0.0036 ± 2.78e-05s
Test IRTraversal.walk_block_ops ran in: 0.014 ± 5.28e-05s
Test Extensibility.interface_check_trait ran in: 2.53e-08 ± 3.66e-07s
Test Extensibility.interface_check ran in: 5.38e-08 ± 3.16e-07s
Test Extensibility.trait_check ran in: 1.22e-06 ± 5.73e-07s
Test Extensibility.trait_check_optimised ran in: 2.24e-07 ± 3.03e-08s
Test Extensibility.trait_check_single ran in: 1.04e-06 ± 5.17e-07s
Test Extensibility.trait_check_neg ran in: 2.11e-06 ± 6.1e-07s
Test OpCreation.operation_create ran in: 2.98e-06 ± 7.59e-07s
Test OpCreation.operation_build ran in: 1.16e-05 ± 1.56e-06s
Test OpCreation.operation_create_optimised ran in: 3.91e-07 ± 3.57e-08s
Test OpCreation.operation_clone ran in: 0.000741 ± 1.11e-05s
Test OpCreation.operation_clone_single ran in: 5.08e-06 ± 1.23e-06s
Test OpCreation.operation_constant_init ran in: 2.85e-05 ± 3.76e-06s
Test OpCreation.operation_constant_create ran in: 1.95e-06 ± 1.31e-06s
Test ConstantFoldingSimple.20 ran in: 0.00094 ± 2.9e-05s
Test ConstantFoldingSimpleInlined.20 ran in: 0.000105 ± 2.55e-05s
```

Listing D.5: Results for the xDSL micro-benchmarks derived from “How Slow is MLIR?”, for CPython version 3.13.3 with the experimental JIT enabled.

Appendix E

Bytecode profiles

```
//// Trace of `OpCreation.time_operation_build` :

// == microbenchmarks:278 `time_operation_build` ==
// >>> EmptyOp.build()
295         0   LOAD_GLOBAL         0   (EmptyOp)                                   // 8    ns
            2   LOAD_METHOD         1   (build)                                     // 10   ns
            4   CALL_METHOD         0   ()                                          // 38   ns

    // ========== operations:160 `build` ===========
    // >>> op = cls.__new__(cls)
    184         0   LOAD_FAST           0   (cls)                                   // 15   ns
                2   LOAD_METHOD         0   (__new__)                               // 15   ns
                4   LOAD_FAST           0   (cls)                                   // 15   ns
                6   CALL_METHOD         1   ()                                      // 18   ns
                8   STORE_FAST          7   (op)                                    // 15   ns
    // >>> IRDLOperation.__init__(
    185         10  LOAD_GLOBAL         1   (IRDLOperation)                         // 13   ns
                12  LOAD_ATTR           2   (__init__)                              // 14   ns
    // >>> op,
    186         14  LOAD_FAST           7   (op)                                    // 12   ns
    // >>> operands=operands,
    187         16  LOAD_FAST           1   (operands)                              // 12   ns
    // >>> result_types=result_types,
    188         18  LOAD_FAST           2   (result_types)                          // 12   ns
    // >>> properties=properties,
    189         20  LOAD_FAST           4   (properties)                            // 12   ns
    // >>> attributes=attributes,
    190         22  LOAD_FAST           3   (attributes)                            // 12   ns
    // >>> successors=successors,
    191         24  LOAD_FAST           5   (successors)                            // 12   ns
    // >>> regions=regions,
    192         26  LOAD_FAST           6   (regions)                               // 12   ns
    // >>> IRDLOperation.__init__(
    185         28  LOAD_CONST          1   (('operands', 'result_types', 'properties', 'attributes',
    ↪  'successors', 'regions'))  // 13   ns
                30  CALL_FUNCTION_KW    7   ()                                      // 33   ns

        // ========= operations:93 `__init__` ==========
        // >>> if operands is None:
        115         0   LOAD_FAST           1   (operands)                          // 12   ns
                    2   LOAD_CONST          0   (None)                              // 12   ns
                    4   IS_OP               0   ()                                  // 11   ns
```

```
                6   POP_JUMP_IF_FALSE    6   (to 12)                              // 12   ns
    // >>> operands = []
    116         8   BUILD_LIST           0   ()                                   // 12   ns
                10  STORE_FAST           1   (operands)                           // 12   ns
    // >>> if result_types is None:
    117     >>  12  LOAD_FAST            2   (result_types)                       // 12   ns
                14  LOAD_CONST           0   (None)                               // 11   ns
                16  IS_OP                0   ()                                   // 12   ns
                18  POP_JUMP_IF_FALSE    12  (to 24)                              // 12   ns
    // >>> result_types = []
    118         20  BUILD_LIST           0   ()                                   // 12   ns
                22  STORE_FAST           2   (result_types)                       // 12   ns
    // >>> if properties is None:
    119     >>  24  LOAD_FAST            3   (properties)                         // 11   ns
                26  LOAD_CONST           0   (None)                               // 12   ns
                28  IS_OP                0   ()                                   // 11   ns
                30  POP_JUMP_IF_FALSE    18  (to 36)                              // 12   ns
    // >>> properties = {}
    120         32  BUILD_MAP            0   ()                                   // 12   ns
                34  STORE_FAST           3   (properties)                         // 12   ns
    // >>> if attributes is None:
    121     >>  36  LOAD_FAST            4   (attributes)                         // 11   ns
                38  LOAD_CONST           0   (None)                               // 11   ns
                40  IS_OP                0   ()                                   // 12   ns
                42  POP_JUMP_IF_FALSE    24  (to 48)                              // 12   ns
    // >>> attributes = {}
    122         44  BUILD_MAP            0   ()                                   // 12   ns
                46  STORE_FAST           4   (attributes)                         // 12   ns
    // >>> if successors is None:
    123     >>  48  LOAD_FAST            5   (successors)                         // 12   ns
                50  LOAD_CONST           0   (None)                               // 11   ns
                52  IS_OP                0   ()                                   // 12   ns
                54  POP_JUMP_IF_FALSE    30  (to 60)                              // 12   ns
    // >>> successors = []
    124         56  BUILD_LIST           0   ()                                   // 12   ns
                58  STORE_FAST           5   (successors)                         // 12   ns
    // >>> if regions is None:
    125     >>  60  LOAD_FAST            6   (regions)                            // 12   ns
                62  LOAD_CONST           0   (None)                               // 11   ns
                64  IS_OP                0   ()                                   // 12   ns
                66  POP_JUMP_IF_FALSE    36  (to 72)                              // 12   ns
    // >>> regions = []
    126         68  BUILD_LIST           0   ()                                   // 12   ns
                70  STORE_FAST           6   (regions)                            // 12   ns
    // >>> irdl_op_init(
    127     >>  72  LOAD_GLOBAL          0   (irdl_op_init)                       // 12   ns
    // >>> self,
    128         74  LOAD_FAST            0   (self)                               // 12   ns
    // >>> type(self).get_irdl_definition(),
    129         76  LOAD_GLOBAL          1   (type)                               // 12   ns
                78  LOAD_FAST            0   (self)                               // 11   ns
                80  CALL_FUNCTION        1   ()                                   // 12   ns
                82  LOAD_METHOD          2   (get_irdl_definition)                // 14   ns
                84  CALL_METHOD          0   ()                                   // 24   ns

        // === operations:2004 `get_irdl_definition` ===
        // >>> return op_def
        2006        0   LOAD_DEREF           0   (op_def)                         // 10   ns
                    2   RETURN_VALUE             ()                               // 24   ns
        // ============================================

    // >>> operands=operands,
    130         86  LOAD_FAST            1   (operands)                           // 12   ns
```

```
        // >>> result_types=result_types,
        131         88  LOAD_FAST           2   (result_types)                          // 12   ns
        // >>> properties=properties,
        132         90  LOAD_FAST           3   (properties)                            // 11   ns
        // >>> attributes=attributes,
        133         92  LOAD_FAST           4   (attributes)                            // 11   ns
        // >>> successors=successors,
        134         94  LOAD_FAST           5   (successors)                            // 11   ns
        // >>> regions=regions,
        135         96  LOAD_FAST           6   (regions)                               // 11   ns
        // >>> irdl_op_init(
        127         98  LOAD_CONST          1   (('operands', 'result_types', 'properties', 'attributes',
        ↪  'successors', 'regions'))  // 11   ns
                    100 CALL_FUNCTION_KW    8   ()                                      // 127  ns

            // ====== operations:1738 `irdl_op_init` =======
            // >>> from xdsl.dialects.builtin import DenseArrayBase, i32
            1760         0   LOAD_CONST          1   (0)                                // 71   ns
                         2   LOAD_CONST          2   (('DenseArrayBase', 'i32'))        // 69   ns
                         4   IMPORT_NAME         0   (xdsl.dialects.builtin)            // 100  ns
                         6   IMPORT_FROM         1   (DenseArrayBase)                   // 69   ns
                         8   STORE_FAST          8   (DenseArrayBase)                   // 68   ns
                         10  IMPORT_FROM         2   (i32)                              // 67   ns
                         12  STORE_FAST          9   (i32)                              // 66   ns
                         14  POP_TOP                 ()                                 // 65   ns
            // >>> error_prefix = f"Error in {op_def.name} builder: "
            1762         16  LOAD_CONST          3   ('Error in ')                      // 64   ns
                         18  LOAD_FAST           1   (op_def)                           // 63   ns
                         20  LOAD_ATTR           3   (name)                             // 64   ns
                         22  FORMAT_VALUE        0   ()                                 // 63   ns
                         24  LOAD_CONST          4   (' builder: ')                     // 62   ns
                         26  BUILD_STRING        3   ()                                 // 65   ns
                         28  STORE_FAST          10  (error_prefix)                     // 67   ns
            // >>> operands_arg = [irdl_build_operations_arg(operand) for operand in operands]
            1764         30  LOAD_CONST          5   (<code object <listcomp> at 0x70c199c29dc0, file
            ↪  "/home/ubuntu/Desktop/xdsl/xdsl/irdl/operations.py", line 1764>)  // 65   ns
                         32  LOAD_CONST          6   ('irdl_op_init.<locals>.<listcomp>')   // 62   ns
                         34  MAKE_FUNCTION       0   ()                                 // 64   ns
                         36  LOAD_FAST           2   (operands)                         // 60   ns
                         38  GET_ITER                ()                                 // 61   ns
                         40  CALL_FUNCTION       1   ()                                 // 48   ns

                // ======= operations:1764 `<listcomp>` ========
                // >>> operands_arg = [irdl_build_operations_arg(operand) for operand in operands]
                1764         0   BUILD_LIST          0   ()                             // 10   ns
                             2   LOAD_FAST           0   (.0)                           // 11   ns
                         >>  4   FOR_ITER            6   (to 18)                        // 11   ns
                         >>  18  RETURN_VALUE            ()                             // 51   ns
                // ============================================

                         42  STORE_FAST          11  (operands_arg)                     // 61   ns
            // >>> regions_arg = [irdl_build_regions_arg(region) for region in regions]
            1766         44  LOAD_CONST          7   (<code object <listcomp> at 0x70c199c29e70, file
            ↪  "/home/ubuntu/Desktop/xdsl/xdsl/irdl/operations.py", line 1766>)  // 59   ns
                         46  LOAD_CONST          6   ('irdl_op_init.<locals>.<listcomp>')   // 58   ns
                         48  MAKE_FUNCTION       0   ()                                 // 60   ns
                         50  LOAD_FAST           7   (regions)                          // 58   ns
                         52  GET_ITER                ()                                 // 58   ns
                         54  CALL_FUNCTION       1   ()                                 // 48   ns

                // ======= operations:1766 `<listcomp>` ========
                // >>> regions_arg = [irdl_build_regions_arg(region) for region in regions]
                1766         0   BUILD_LIST          0   ()                             // 11   ns
```

```
                2   LOAD_FAST           0   (.0)                             // 11   ns
            >>  4   FOR_ITER            6   (to 18)                          // 11   ns
            >>  18  RETURN_VALUE            ()                               // 50   ns
    // =============================================

            56  STORE_FAST          12  (regions_arg)                        // 60   ns
// >>> built_operands, operand_sizes = irdl_build_arg_list(
1769         58  LOAD_GLOBAL         4   (irdl_build_arg_list)               // 60   ns
// >>> VarIRConstruct.OPERAND, operands_arg, op_def.operands, error_prefix
1770         60  LOAD_GLOBAL         5   (VarIRConstruct)                    // 59   ns
            62  LOAD_ATTR           6   (OPERAND)                            // 65   ns
            64  LOAD_FAST           11  (operands_arg)                       // 58   ns
            66  LOAD_FAST           1   (op_def)                             // 57   ns
            68  LOAD_ATTR           7   (operands)                           // 57   ns
            70  LOAD_FAST           10  (error_prefix)                       // 57   ns
// >>> built_operands, operand_sizes = irdl_build_arg_list(
1769         72  CALL_FUNCTION       4   ()                                  // 70   ns

    // === operations:1637 `irdl_build_arg_list` ===
    // >>> if len(args) != len(arg_defs):
    1645        0   LOAD_GLOBAL         0   (len)                            // 27   ns
                2   LOAD_FAST           1   (args)                           // 25   ns
                4   CALL_FUNCTION       1   ()                               // 26   ns
                6   LOAD_GLOBAL         0   (len)                            // 24   ns
                8   LOAD_FAST           2   (arg_defs)                       // 23   ns
                10  CALL_FUNCTION       1   ()                               // 24   ns
                12  COMPARE_OP          3   (!=)                             // 24   ns
                14  POP_JUMP_IF_FALSE   27  (to 54)                          // 24   ns
    // >>> res = list[_T]()
    1651    >>  54  LOAD_GLOBAL         3   (list)                           // 23   ns
                56  LOAD_GLOBAL         4   (_T)                             // 23   ns
                58  BINARY_SUBSCR           ()                               // 28   ns
                60  CALL_FUNCTION       0   ()                               // 37   ns
                62  STORE_FAST          4   (res)                            // 25   ns
    // >>> arg_sizes = list[int]()
    1652        64  LOAD_GLOBAL         3   (list)                           // 23   ns
                66  LOAD_GLOBAL         5   (int)                            // 22   ns
                68  BINARY_SUBSCR           ()                               // 26   ns
                70  CALL_FUNCTION       0   ()                               // 33   ns
                72  STORE_FAST          5   (arg_sizes)                      // 26   ns
    // >>> for arg_idx, ((arg_name, arg_def), arg) in enumerate(zip(arg_defs, args)):
    1654        74  LOAD_GLOBAL         6   (enumerate)                      // 25   ns
                76  LOAD_GLOBAL         7   (zip)                            // 23   ns
                78  LOAD_FAST           2   (arg_defs)                       // 23   ns
                80  LOAD_FAST           1   (args)                           // 21   ns
                82  CALL_FUNCTION       2   ()                               // 24   ns
                84  CALL_FUNCTION       1   ()                               // 23   ns
                86  GET_ITER                ()                               // 21   ns
            >>  88  FOR_ITER            126 (to 342)                         // 26   ns
    // >>> return res, arg_sizes
    1683    >>  342 LOAD_FAST           4   (res)                            // 20   ns
                344 LOAD_FAST           5   (arg_sizes)                      // 20   ns
                346 BUILD_TUPLE         2   ()                               // 20   ns
                348 RETURN_VALUE            ()                               // 65   ns
    // =============================================

            74  UNPACK_SEQUENCE     2   ()                                   // 60   ns
            76  STORE_FAST          13  (built_operands)                     // 59   ns
            78  STORE_FAST          14  (operand_sizes)                      // 58   ns
// >>> built_res_types, result_sizes = irdl_build_arg_list(
1774         80  LOAD_GLOBAL         4   (irdl_build_arg_list)               // 57   ns
// >>> VarIRConstruct.RESULT, result_types, op_def.results, error_prefix
1775         82  LOAD_GLOBAL         5   (VarIRConstruct)                    // 55   ns
```

```
                84  LOAD_ATTR           8   (RESULT)                            // 59   ns
                86  LOAD_FAST           3   (result_types)                      // 54   ns
                88  LOAD_FAST           1   (op_def)                            // 53   ns
                90  LOAD_ATTR           9   (results)                           // 53   ns
                92  LOAD_FAST           10  (error_prefix)                      // 53   ns
        // >>> built_res_types, result_sizes = irdl_build_arg_list(
        1774         94  CALL_FUNCTION       4   ()                             // 69   ns

            // === operations:1637 `irdl_build_arg_list` ===
            // >>> if len(args) != len(arg_defs):
            1645         0   LOAD_GLOBAL         0   (len)                      // 26   ns
                         2   LOAD_FAST           1   (args)                     // 26   ns
                         4   CALL_FUNCTION       1   ()                         // 26   ns
                         6   LOAD_GLOBAL         0   (len)                      // 24   ns
                         8   LOAD_FAST           2   (arg_defs)                 // 23   ns
                         10  CALL_FUNCTION       1   ()                         // 24   ns
                         12  COMPARE_OP          3   (!=)                       // 24   ns
                         14  POP_JUMP_IF_FALSE   27  (to 54)                    // 24   ns
            // >>> res = list[_T]()
            1651     >>  54  LOAD_GLOBAL         3   (list)                     // 23   ns
                         56  LOAD_GLOBAL         4   (_T)                       // 23   ns
                         58  BINARY_SUBSCR           ()                         // 27   ns
                         60  CALL_FUNCTION       0   ()                         // 35   ns
                         62  STORE_FAST          4   (res)                      // 25   ns
            // >>> arg_sizes = list[int]()
            1652         64  LOAD_GLOBAL         3   (list)                     // 23   ns
                         66  LOAD_GLOBAL         5   (int)                      // 23   ns
                         68  BINARY_SUBSCR           ()                         // 26   ns
                         70  CALL_FUNCTION       0   ()                         // 33   ns
                         72  STORE_FAST          5   (arg_sizes)                // 25   ns
            // >>> for arg_idx, ((arg_name, arg_def), arg) in enumerate(zip(arg_defs, args)):
            1654         74  LOAD_GLOBAL         6   (enumerate)                // 23   ns
                         76  LOAD_GLOBAL         7   (zip)                      // 22   ns
                         78  LOAD_FAST           2   (arg_defs)                 // 22   ns
                         80  LOAD_FAST           1   (args)                     // 20   ns
                         82  CALL_FUNCTION       2   ()                         // 24   ns
                         84  CALL_FUNCTION       1   ()                         // 23   ns
                         86  GET_ITER                ()                         // 21   ns
                     >>  88  FOR_ITER            126 (to 342)                   // 25   ns
            // >>> return res, arg_sizes
            1683     >>  342 LOAD_FAST           4   (res)                      // 20   ns
                         344 LOAD_FAST           5   (arg_sizes)                // 20   ns
                         346 BUILD_TUPLE         2   ()                         // 20   ns
                         348 RETURN_VALUE            ()                         // 63   ns
            // =============================================

                96  UNPACK_SEQUENCE     2   ()                                  // 56   ns
                98  STORE_FAST          15  (built_res_types)                   // 56   ns
                100 STORE_FAST          16  (result_sizes)                      // 54   ns
        // >>> built_regions, region_sizes = irdl_build_arg_list(
        1779         102 LOAD_GLOBAL         4   (irdl_build_arg_list)          // 53   ns
        // >>> VarIRConstruct.REGION, regions_arg, op_def.regions, error_prefix
        1780         104 LOAD_GLOBAL         5   (VarIRConstruct)               // 51   ns
                106 LOAD_ATTR           10  (REGION)                            // 55   ns
                108 LOAD_FAST           12  (regions_arg)                       // 51   ns
                110 LOAD_FAST           1   (op_def)                            // 49   ns
                112 LOAD_ATTR           11  (regions)                           // 50   ns
                114 LOAD_FAST           10  (error_prefix)                      // 49   ns
        // >>> built_regions, region_sizes = irdl_build_arg_list(
        1779         116 CALL_FUNCTION       4   ()                             // 67   ns

            // === operations:1637 `irdl_build_arg_list` ===
            // >>> if len(args) != len(arg_defs):
```

```
    1645         0   LOAD_GLOBAL          0   (len)                          // 26   ns
                 2   LOAD_FAST            1   (args)                         // 26   ns
                 4   CALL_FUNCTION        1   ()                             // 26   ns
                 6   LOAD_GLOBAL          0   (len)                          // 23   ns
                 8   LOAD_FAST            2   (arg_defs)                     // 24   ns
                 10  CALL_FUNCTION        1   ()                             // 24   ns
                 12  COMPARE_OP           3   (!=)                           // 24   ns
                 14  POP_JUMP_IF_FALSE    27  (to 54)                        // 24   ns
    // >>> res = list[_T]()
    1651     >>  54  LOAD_GLOBAL          3   (list)                         // 23   ns
                 56  LOAD_GLOBAL          4   (_T)                           // 23   ns
                 58  BINARY_SUBSCR            ()                             // 27   ns
                 60  CALL_FUNCTION        0   ()                             // 35   ns
                 62  STORE_FAST           4   (res)                          // 25   ns
    // >>> arg_sizes = list[int]()
    1652         64  LOAD_GLOBAL          3   (list)                         // 24   ns
                 66  LOAD_GLOBAL          5   (int)                          // 23   ns
                 68  BINARY_SUBSCR            ()                             // 26   ns
                 70  CALL_FUNCTION        0   ()                             // 33   ns
                 72  STORE_FAST           5   (arg_sizes)                    // 25   ns
    // >>> for arg_idx, ((arg_name, arg_def), arg) in enumerate(zip(arg_defs, args)):
    1654         74  LOAD_GLOBAL          6   (enumerate)                    // 23   ns
                 76  LOAD_GLOBAL          7   (zip)                          // 22   ns
                 78  LOAD_FAST            2   (arg_defs)                     // 21   ns
                 80  LOAD_FAST            1   (args)                         // 20   ns
                 82  CALL_FUNCTION        2   ()                             // 24   ns
                 84  CALL_FUNCTION        1   ()                             // 23   ns
                 86  GET_ITER                 ()                             // 21   ns
             >>  88  FOR_ITER             126 (to 342)                       // 25   ns
    // >>> return res, arg_sizes
    1683     >>  342 LOAD_FAST            4   (res)                          // 20   ns
                 344 LOAD_FAST            5   (arg_sizes)                    // 20   ns
                 346 BUILD_TUPLE          2   ()                             // 20   ns
                 348 RETURN_VALUE             ()                             // 61   ns
    // =============================================

             118 UNPACK_SEQUENCE      2   ()                                 // 52   ns
             120 STORE_FAST           17  (built_regions)                    // 51   ns
             122 STORE_FAST           18  (region_sizes)                     // 50   ns
// >>> built_successors, successor_sizes = irdl_build_arg_list(
1784         124 LOAD_GLOBAL          4   (irdl_build_arg_list)              // 50   ns
// >>> VarIRConstruct.SUCCESSOR, successors, op_def.successors, error_prefix
1785         126 LOAD_GLOBAL          5   (VarIRConstruct)                   // 48   ns
             128 LOAD_ATTR            12  (SUCCESSOR)                        // 52   ns
             130 LOAD_FAST            6   (successors)                       // 47   ns
             132 LOAD_FAST            1   (op_def)                           // 46   ns
             134 LOAD_ATTR            13  (successors)                       // 47   ns
             136 LOAD_FAST            10  (error_prefix)                     // 46   ns
// >>> built_successors, successor_sizes = irdl_build_arg_list(
1784         138 CALL_FUNCTION        4   ()                                 // 65   ns

    // === operations:1637 `irdl_build_arg_list` ===
    // >>> if len(args) != len(arg_defs):
    1645         0   LOAD_GLOBAL          0   (len)                          // 26   ns
                 2   LOAD_FAST            1   (args)                         // 26   ns
                 4   CALL_FUNCTION        1   ()                             // 26   ns
                 6   LOAD_GLOBAL          0   (len)                          // 24   ns
                 8   LOAD_FAST            2   (arg_defs)                     // 24   ns
                 10  CALL_FUNCTION        1   ()                             // 24   ns
                 12  COMPARE_OP           3   (!=)                           // 23   ns
                 14  POP_JUMP_IF_FALSE    27  (to 54)                        // 23   ns
    // >>> res = list[_T]()
    1651     >>  54  LOAD_GLOBAL          3   (list)                         // 23   ns
```

```
                 56  LOAD_GLOBAL         4   (_T)                                    // 23   ns
                 58  BINARY_SUBSCR           ()                                      // 27   ns
                 60  CALL_FUNCTION       0   ()                                      // 35   ns
                 62  STORE_FAST          4   (res)                                   // 25   ns
     // >>> arg_sizes = list[int]()
     1652         64  LOAD_GLOBAL         3   (list)                                 // 23   ns
                 66  LOAD_GLOBAL         5   (int)                                   // 23   ns
                 68  BINARY_SUBSCR           ()                                      // 26   ns
                 70  CALL_FUNCTION       0   ()                                      // 33   ns
                 72  STORE_FAST          5   (arg_sizes)                             // 25   ns
     // >>> for arg_idx, ((arg_name, arg_def), arg) in enumerate(zip(arg_defs, args)):
     1654         74  LOAD_GLOBAL         6   (enumerate)                            // 23   ns
                 76  LOAD_GLOBAL         7   (zip)                                   // 22   ns
                 78  LOAD_FAST           2   (arg_defs)                              // 21   ns
                 80  LOAD_FAST           1   (args)                                  // 20   ns
                 82  CALL_FUNCTION       2   ()                                      // 23   ns
                 84  CALL_FUNCTION       1   ()                                      // 23   ns
                 86  GET_ITER                ()                                      // 21   ns
             >>  88  FOR_ITER            126 (to 342)                                // 25   ns
     // >>> return res, arg_sizes
     1683     >>  342 LOAD_FAST           4   (res)                                  // 20   ns
                 344 LOAD_FAST           5   (arg_sizes)                             // 20   ns
                 346 BUILD_TUPLE         2   ()                                      // 20   ns
                 348 RETURN_VALUE            ()                                      // 59   ns
     // =============================================

             140 UNPACK_SEQUENCE     2   ()                                          // 49   ns
             142 STORE_FAST          19  (built_successors)                          // 48   ns
             144 STORE_FAST          20  (successor_sizes)                           // 46   ns
 // >>> built_properties = dict[str, Attribute]()
 1789         146 LOAD_GLOBAL         14  (dict)                                     // 45   ns
             148 LOAD_GLOBAL         15  (str)                                       // 43   ns
             150 LOAD_GLOBAL         16  (Attribute)                                 // 42   ns
             152 BUILD_TUPLE         2   ()                                          // 43   ns
             154 BINARY_SUBSCR           ()                                          // 46   ns
             156 CALL_FUNCTION       0   ()                                          // 54   ns
             158 STORE_FAST          21  (built_properties)                          // 49   ns
 // >>> for attr_name, attr in properties.items():
 1790         160 LOAD_FAST           4   (properties)                               // 45   ns
             162 LOAD_METHOD         17  (items)                                     // 44   ns
             164 CALL_METHOD         0   ()                                          // 45   ns
             166 GET_ITER                ()                                          // 43   ns
         >>  168 FOR_ITER            13  (to 196)                                    // 42   ns
 // >>> built_attributes = dict[str, Attribute]()
 1796     >>  196 LOAD_GLOBAL         14  (dict)                                     // 40   ns
             198 LOAD_GLOBAL         15  (str)                                       // 39   ns
             200 LOAD_GLOBAL         16  (Attribute)                                 // 40   ns
             202 BUILD_TUPLE         2   ()                                          // 41   ns
             204 BINARY_SUBSCR           ()                                          // 43   ns
             206 CALL_FUNCTION       0   ()                                          // 51   ns
             208 STORE_FAST          24  (built_attributes)                          // 41   ns
 // >>> for attr_name, attr in attributes.items():
 1797         210 LOAD_FAST           5   (attributes)                               // 39   ns
             212 LOAD_METHOD         17  (items)                                     // 39   ns
             214 CALL_METHOD         0   ()                                          // 40   ns
             216 GET_ITER                ()                                          // 39   ns
         >>  218 FOR_ITER            13  (to 246)                                    // 39   ns
 // >>> for option in op_def.options:
 1803     >>  246 LOAD_FAST           1   (op_def)                                   // 38   ns
             248 LOAD_ATTR           18  (options)                                   // 38   ns
             250 GET_ITER                ()                                          // 38   ns
         >>  252 FOR_ITER            223 (to 700)                                    // 43   ns
 // >>> Operation.__init__(
```

```
1863     >>  700 LOAD_GLOBAL         37  (Operation)                       // 38   ns
            702 LOAD_ATTR           38  (__init__)                        // 40   ns
// >>> self,
1864         704 LOAD_FAST           0   (self)                            // 38   ns
// >>> operands=built_operands,
1865         706 LOAD_FAST           13  (built_operands)                  // 37   ns
// >>> result_types=built_res_types,
1866         708 LOAD_FAST           15  (built_res_types)                 // 37   ns
// >>> properties=built_properties,
1867         710 LOAD_FAST           21  (built_properties)                // 37   ns
// >>> attributes=built_attributes,
1868         712 LOAD_FAST           24  (built_attributes)                // 37   ns
// >>> successors=built_successors,
1869         714 LOAD_FAST           19  (built_successors)                // 37   ns
// >>> regions=built_regions,
1870         716 LOAD_FAST           17  (built_regions)                   // 37   ns
// >>> Operation.__init__(
1863         718 LOAD_CONST          19  (('operands', 'result_types', 'properties',
↪  'attributes', 'successors', 'regions'))  // 37   ns
            720 CALL_FUNCTION_KW     7  ()                                // 55   ns

    // ============ core:914 `__init__` ============
    // >>> super().__init__()
    924         0   LOAD_GLOBAL         0   (super)                       // 18   ns
                2   CALL_FUNCTION       0   ()                            // 21   ns
                4   LOAD_METHOD         1   (__init__)                    // 22   ns
                6   CALL_METHOD         0   ()                            // 20   ns
                8   POP_TOP                 ()                            // 17   ns
    // >>> self.operands = operands
    927         10  LOAD_FAST           1   (operands)                    // 16   ns
                12  LOAD_DEREF          0   (self)                        // 16   ns
                14  STORE_ATTR          2   (operands)                    // 36   ns

        // ============ core:888 `operands` ============
        // >>> new = tuple(new)
        890         0   LOAD_GLOBAL         0   (tuple)                   // 15   ns
                    2   LOAD_FAST           1   (new)                     // 14   ns
                    4   CALL_FUNCTION       1   ()                        // 15   ns
                    6   STORE_FAST          1   (new)                     // 14   ns
        // >>> for idx, operand in enumerate(self._operands):
        891         8   LOAD_GLOBAL         1   (enumerate)               // 13   ns
                    10  LOAD_FAST           0   (self)                    // 13   ns
                    12  LOAD_ATTR           2   (_operands)               // 13   ns
                    14  CALL_FUNCTION       1   ()                        // 16   ns
                    16  GET_ITER                ()                        // 14   ns
                >>  18  FOR_ITER            12  (to 44)                   // 15   ns
        // >>> for idx, operand in enumerate(new):
        893     >>  44  LOAD_GLOBAL         1   (enumerate)               // 13   ns
                    46  LOAD_FAST           1   (new)                     // 14   ns
                    48  CALL_FUNCTION       1   ()                        // 15   ns
                    50  GET_ITER                ()                        // 13   ns
                >>  52  FOR_ITER            12  (to 78)                   // 15   ns
        // >>> self._operands = new
        895     >>  78  LOAD_FAST           1   (new)                     // 13   ns
                    80  LOAD_FAST           0   (self)                    // 13   ns
                    82  STORE_ATTR          2   (_operands)               // 15   ns
                    84  LOAD_CONST          0   (None)                    // 13   ns
                    86  RETURN_VALUE            ()                        // 32   ns
        // ============================================

    // >>> self.results = tuple(
    929         16  LOAD_GLOBAL         3   (tuple)                       // 17   ns
                18  LOAD_CLOSURE        0   (self)                        // 17   ns
```

```
                20  BUILD_TUPLE         1   ()                                      // 17   ns
                22  LOAD_CONST          1   (<code object <genexpr> at 0x70c199e02b80, file
                ↪   "/home/ubuntu/Desktop/xdsl/xdsl/ir/core.py", line 929>)  // 17   ns
                24  LOAD_CONST          2   ('Operation.__init__.<locals>.<genexpr>')  // 16
                ↪   ns
                26  MAKE_FUNCTION       8   (closure)                               // 18   ns
// >>> for (idx, result_type) in enumerate(result_types)
931             28  LOAD_GLOBAL         4   (enumerate)                             // 17   ns
                30  LOAD_FAST           2   (result_types)                          // 16   ns
                32  CALL_FUNCTION       1   ()                                      // 20   ns
// >>> self.results = tuple(
929             34  GET_ITER                ()                                      // 17   ns
                36  CALL_FUNCTION       1   ()                                      // 20   ns
                38  CALL_FUNCTION       1   ()                                      // 36   ns

    // =========== core:929 `<genexpr>` ============
                    0   GEN_START           0   ()                                  // 15   ns
    // >>> self.results = tuple(
    929             2   LOAD_FAST           0   (.0)                                // 15   ns
                >>  4   FOR_ITER            11  (to 28)                             // 15   ns
                >>  28  LOAD_CONST          0   (None)                              // 14   ns
                    30  RETURN_VALUE            ()                                  // 38   ns
    // ============================================

                40  LOAD_DEREF          0   (self)                                  // 17   ns
                42  STORE_ATTR          5   (results)                               // 18   ns
// >>> self.properties = dict(properties)
933             44  LOAD_GLOBAL         6   (dict)                                  // 17   ns
                46  LOAD_FAST           3   (properties)                            // 17   ns
                48  CALL_FUNCTION       1   ()                                      // 19   ns
                50  LOAD_DEREF          0   (self)                                  // 17   ns
                52  STORE_ATTR          7   (properties)                            // 18   ns
// >>> self.attributes = dict(attributes)
934             54  LOAD_GLOBAL         6   (dict)                                  // 16   ns
                56  LOAD_FAST           4   (attributes)                            // 16   ns
                58  CALL_FUNCTION       1   ()                                      // 18   ns
                60  LOAD_DEREF          0   (self)                                  // 17   ns
                62  STORE_ATTR          8   (attributes)                            // 17   ns
// >>> self.successors = list(successors)
935             64  LOAD_GLOBAL         9   (list)                                  // 16   ns
                66  LOAD_FAST           5   (successors)                            // 16   ns
                68  CALL_FUNCTION       1   ()                                      // 18   ns
                70  LOAD_DEREF          0   (self)                                  // 17   ns
                72  STORE_ATTR          10  (successors)                            // 33   ns

    // =========== core:901 `successors` ===========
    // >>> new = tuple(new)
    903             0   LOAD_GLOBAL         0   (tuple)                             // 14   ns
                    2   LOAD_FAST           1   (new)                               // 13   ns
                    4   CALL_FUNCTION       1   ()                                  // 15   ns
                    6   STORE_FAST          1   (new)                               // 13   ns
    // >>> for idx, successor in enumerate(self._successors):
    904             8   LOAD_GLOBAL         1   (enumerate)                         // 13   ns
                    10  LOAD_FAST           0   (self)                              // 13   ns
                    12  LOAD_ATTR           2   (_successors)                       // 14   ns
                    14  CALL_FUNCTION       1   ()                                  // 16   ns
                    16  GET_ITER                ()                                  // 14   ns
                >>  18  FOR_ITER            12  (to 44)                             // 15   ns
    // >>> for idx, successor in enumerate(new):
    906         >>  44  LOAD_GLOBAL         1   (enumerate)                         // 13   ns
                    46  LOAD_FAST           1   (new)                               // 13   ns
                    48  CALL_FUNCTION       1   ()                                  // 15   ns
                    50  GET_ITER                ()                                  // 13   ns
```

```
                >>  52  FOR_ITER           12  (to 78)                      // 15   ns
        // >>> self._successors = new
        908     >>  78  LOAD_FAST          1   (new)                        // 13   ns
                    80  LOAD_FAST          0   (self)                       // 13   ns
                    82  STORE_ATTR         2   (_successors)                // 14   ns
                    84  LOAD_CONST         0   (None)                       // 13   ns
                    86  RETURN_VALUE           ()                           // 32   ns
        // =============================================

    // >>> self.regions = ()
    936         74  LOAD_CONST         3   (())                             // 17   ns
                76  LOAD_DEREF         0   (self)                           // 16   ns
                78  STORE_ATTR         11  (regions)                        // 20   ns
    // >>> for region in regions:
    937         80  LOAD_FAST          6   (regions)                        // 17   ns
                82  GET_ITER               ()                               // 17   ns
            >>  84  FOR_ITER           7   (to 100)                         // 17   ns
    // >>> self.__post_init__()
    940     >>  100 LOAD_DEREF         0   (self)                           // 17   ns
                102 LOAD_METHOD        13  (__post_init__)                  // 17   ns
                104 CALL_METHOD        0   ()                               // 45   ns

        // ====== operations:138 `__post_init__` =======
        // >>> op_def = self.get_irdl_definition()
        139         0   LOAD_FAST          0   (self)                       // 22   ns
                    2   LOAD_METHOD        0   (get_irdl_definition)        // 24   ns
                    4   CALL_METHOD        0   ()                           // 28   ns

            // === operations:2004 `get_irdl_definition` ===
            // >>> return op_def
            2006         0   LOAD_DEREF          0   (op_def)               // 10   ns
                        2   RETURN_VALUE            ()                      // 28   ns
            // =============================================

                    6   STORE_FAST         1   (op_def)                     // 21   ns
        // >>> for prop_name, prop_def in op_def.properties.items():
        141         8   LOAD_FAST          1   (op_def)                     // 21   ns
                    10  LOAD_ATTR          1   (properties)                 // 21   ns
                    12  LOAD_METHOD        2   (items)                      // 20   ns
                    14  CALL_METHOD        0   ()                           // 22   ns
                    16  GET_ITER               ()                           // 21   ns
                >>  18  FOR_ITER           25  (to 70)                      // 21   ns
        // >>> for attr_name, attr_def in op_def.attributes.items():
        150     >>  70  LOAD_FAST          1   (op_def)                     // 20   ns
                    72  LOAD_ATTR          6   (attributes)                 // 19   ns
                    74  LOAD_METHOD        2   (items)                      // 19   ns
                    76  CALL_METHOD        0   ()                           // 21   ns
                    78  GET_ITER               ()                           // 20   ns
                >>  80  FOR_ITER           25  (to 132)                     // 20   ns
        // >>> return super().__post_init__()
        158     >>  132 LOAD_GLOBAL        7   (super)                      // 20   ns
                    134 CALL_FUNCTION      0   ()                           // 22   ns
                    136 LOAD_METHOD        8   (__post_init__)              // 22   ns
                    138 CALL_METHOD        0   ()                           // 27   ns

            // ======== builder:343 `new_post_init` ========
            // >>> old_post_init(self)
            344         0   LOAD_DEREF         0   (old_post_init)          // 9    ns
                        2   LOAD_FAST          0   (self)                   // 9    ns
                        4   CALL_FUNCTION      1   ()                       // 20   ns

                // ========= core:910 `__post_init__` ==========
                // >>> assert self.name != ""
```

```
    911         0   LOAD_FAST            0   (self)                 // 8    ns
                2   LOAD_ATTR            0   (name)                 // 9    ns
                4   LOAD_CONST           1   ('')                   // 8    ns
                6   COMPARE_OP           3   (!=)                   // 8    ns
                8   POP_JUMP_IF_TRUE     7   (to 14)                // 8    ns
    // >>> assert isinstance(self.name, str)
    912     >>  14  LOAD_GLOBAL          1   (isinstance)           // 7    ns
                16  LOAD_FAST            0   (self)                 // 7    ns
                18  LOAD_ATTR            0   (name)                 // 8    ns
                20  LOAD_GLOBAL          2   (str)                  // 7    ns
                22  CALL_FUNCTION        2   ()                     // 9    ns
                24  POP_JUMP_IF_TRUE     15  (to 30)                // 8    ns
            >>  30  LOAD_CONST           0   (None)                 // 7    ns
                32  RETURN_VALUE             ()                     // 18   ns
    // =============================================

            6   POP_TOP                  ()                         // 10   ns
// >>> _op_init_callback(self)
345         8   LOAD_GLOBAL          0   (_op_init_callback)        // 9    ns
            10  LOAD_FAST            0   (self)                     // 9    ns
            12  CALL_FUNCTION        1   ()                         // 24   ns

    // ====== builder:335 `_op_init_callback` ======
    // >>> if (b := ImplicitBuilder.get()) is not None:
    336         0   LOAD_GLOBAL          0   (ImplicitBuilder)      // 11   ns
                2   LOAD_METHOD          1   (get)                  // 13   ns
                4   CALL_METHOD          0   ()                     // 21   ns

        // ============= builder:321 `get` =============
        // >>> return cls._stack.get()
        326         0   LOAD_FAST            0   (cls)              // 7    ns
                    2   LOAD_ATTR            0   (_stack)           // 9    ns
                    4   LOAD_METHOD          1   (get)              // 12   ns
                    6   CALL_METHOD          0   ()                 // 18   ns

            // ============= builder:261 `get` =============
            // >>> if len(self.stack):
            262         0   LOAD_GLOBAL          0   (len)          // 8    ns
                        2   LOAD_FAST            0   (self)         // 7    ns
                        4   LOAD_ATTR            1   (stack)        // 10   ns
                        6   CALL_FUNCTION        1   ()             // 9    ns
                        8   POP_JUMP_IF_FALSE    10  (to 20)        // 9    ns
            262     >>  20  LOAD_CONST           0   (None)         // 8    ns
                        22  RETURN_VALUE             ()             // 18   ns
            // =============================================

                    8   RETURN_VALUE             ()                 // 19   ns
        // =============================================

                6   DUP_TOP                  ()                     // 11   ns
                8   STORE_FAST           1   (b)                    // 10   ns
                10  LOAD_CONST           0   (None)                 // 9    ns
                12  IS_OP                1   ()                     // 9    ns
                14  POP_JUMP_IF_FALSE    15  (to 30)                // 9    ns
    336     >>  30  LOAD_CONST           0   (None)                 // 8    ns
                32  RETURN_VALUE             ()                     // 20   ns
    // =============================================

            14  POP_TOP                  ()                         // 10   ns
            16  LOAD_CONST           0   (None)                     // 9    ns
            18  RETURN_VALUE             ()                         // 27   ns
// =============================================
```

```
                                140 RETURN_VALUE             ()                             // 42   ns
                    // =============================================

                            106 POP_TOP                  ()                                 // 18   ns
                            108 LOAD_CONST           0   (None)                             // 17   ns
                            110 RETURN_VALUE             ()                                 // 51   ns
                // =============================================

                        722 POP_TOP                  ()                                     // 39   ns
                        724 LOAD_CONST           8   (None)                                 // 38   ns
                        726 RETURN_VALUE             ()                                     // 76   ns
            // =============================================

                    102 POP_TOP                  ()                                         // 14   ns
                    104 LOAD_CONST           0   (None)                                     // 13   ns
                    106 RETURN_VALUE             ()                                         // 34   ns
        // =============================================

                32  POP_TOP                  ()                                             // 14   ns
    // >>> return op
    194         34  LOAD_FAST            7   (op)                                           // 13   ns
                36  RETURN_VALUE             ()                                             // 29   ns
    // =============================================

            6   POP_TOP                  ()                                                 // 12   ns
            8   LOAD_CONST           1   (None)                                             // 8    ns
            10  RETURN_VALUE             ()                                                 // 20   ns
// =============================================
```

Listing E.1: Bytecode profile trace of the original implementation of instantiation of an empty operation.

```
//// Trace of `OpCreation.time_operation_create_optimised` :

// == microbenchmarks:297 `time_operation_create_optimised` ==
// >>> empty_op = EmptyOp.__new__(EmptyOp)
299         0   LOAD_GLOBAL          0   (EmptyOp)                                          // 23   ns
            2   LOAD_METHOD          1   (__new__)                                          // 24   ns
            4   LOAD_GLOBAL          0   (EmptyOp)                                          // 22   ns
            6   CALL_METHOD          1   ()                                                 // 26   ns
            8   STORE_FAST           1   (empty_op)                                         // 20   ns
// >>> empty_op._operands = tuple()  # pyright: ignore[reportPrivateUsage]
300         10  LOAD_GLOBAL          2   (tuple)                                            // 19   ns
            12  CALL_FUNCTION        0   ()                                                 // 20   ns
            14  LOAD_FAST            1   (empty_op)                                         // 19   ns
            16  STORE_ATTR           3   (_operands)                                        // 22   ns
// >>> empty_op.results = tuple()
301         18  LOAD_GLOBAL          2   (tuple)                                            // 19   ns
            20  CALL_FUNCTION        0   ()                                                 // 20   ns
            22  LOAD_FAST            1   (empty_op)                                         // 18   ns
            24  STORE_ATTR           4   (results)                                          // 21   ns
// >>> empty_op.properties = {}
302         26  BUILD_MAP            0   ()                                                 // 19   ns
            28  LOAD_FAST            1   (empty_op)                                         // 17   ns
            30  STORE_ATTR           5   (properties)                                       // 21   ns
// >>> empty_op.attributes = {}
303         32  BUILD_MAP            0   ()                                                 // 19   ns
            34  LOAD_FAST            1   (empty_op)                                         // 18   ns
            36  STORE_ATTR           6   (attributes)                                       // 22   ns
// >>> empty_op._successors = tuple()  # pyright: ignore[reportPrivateUsage]
304         38  LOAD_GLOBAL          2   (tuple)                                            // 18   ns
            40  CALL_FUNCTION        0   ()                                                 // 20   ns
```

```
            42  LOAD_FAST           1   (empty_op)                                     // 18   ns
            44  STORE_ATTR          7   (_successors)                                  // 21   ns
// >>> empty_op.regions = tuple()
305         46  LOAD_GLOBAL         2   (tuple)                                        // 18   ns
            48  CALL_FUNCTION       0   ()                                             // 20   ns
            50  LOAD_FAST           1   (empty_op)                                     // 19   ns
            52  STORE_ATTR          8   (regions)                                      // 23   ns
            54  LOAD_CONST          1   (None)                                         // 18   ns
            56  RETURN_VALUE            ()                                             // 44   ns
// =============================================
```

Listing E.2: Bytecode profile trace of the optimised implementation of instantiation an empty operation.

```
//// Trace of `Extensibility.time_trait_check` :

// == microbenchmarks:176 `time_trait_check` ===
// >>> assert Extensibility.OP_WITH_REGION.has_trait(Extensibility.TRAIT_4)
204         0   LOAD_GLOBAL         0   (Extensibility)                                // 17   ns
            2   LOAD_ATTR           1   (OP_WITH_REGION)                               // 18   ns
            4   LOAD_METHOD         2   (has_trait)                                    // 18   ns
            6   LOAD_GLOBAL         0   (Extensibility)                                // 17   ns
            8   LOAD_ATTR           3   (TRAIT_4)                                      // 18   ns
            10  CALL_METHOD         1   ()                                             // 54   ns

    // =========== core:1192 `has_trait` ===========
    // >>> from xdsl.dialects.builtin import UnregisteredOp
    1204        0   LOAD_CONST          1   (0)                                        // 26   ns
                2   LOAD_CONST          2   (('UnregisteredOp',))                      // 25   ns
                4   IMPORT_NAME         0   (xdsl.dialects.builtin)                    // 72   ns
                6   IMPORT_FROM         1   (UnregisteredOp)                           // 28   ns
                8   STORE_FAST          3   (UnregisteredOp)                           // 23   ns
                10  POP_TOP                 ()                                         // 21   ns
    // >>> if issubclass(cls, UnregisteredOp):
    1206        12  LOAD_GLOBAL         2   (issubclass)                               // 21   ns
                14  LOAD_FAST           0   (cls)                                      // 21   ns
                16  LOAD_FAST           3   (UnregisteredOp)                           // 19   ns
                18  CALL_FUNCTION       2   ()                                         // 46   ns

        // ======== abc:121 `__subclasscheck__` ========
        // >>> return _abc_subclasscheck(cls, subclass)
        123         0   LOAD_GLOBAL         0   (_abc_subclasscheck)                   // 19   ns
                    2   LOAD_FAST           0   (cls)                                  // 18   ns
                    4   LOAD_FAST           1   (subclass)                             // 17   ns
                    6   CALL_FUNCTION       2   ()                                     // 26   ns
                    8   RETURN_VALUE            ()                                     // 44   ns
        // =============================================

                20  POP_JUMP_IF_FALSE   13  (to 26)                                    // 22   ns
    // >>> return cls.get_trait(trait) is not None
    1209     >> 26  LOAD_FAST           0   (cls)                                      // 21   ns
                28  LOAD_METHOD         3   (get_trait)                                // 23   ns
                30  LOAD_FAST           1   (trait)                                    // 21   ns
                32  CALL_METHOD         1   ()                                         // 52   ns

        // =========== core:1211 `get_trait` ===========
        // >>> if isinstance(trait, type):
        1216        0   LOAD_GLOBAL         0   (isinstance)                           // 24   ns
                    2   LOAD_FAST           1   (trait)                                // 23   ns
                    4   LOAD_GLOBAL         1   (type)                                 // 25   ns
                    6   CALL_FUNCTION       2   ()                                     // 25   ns
                    8   POP_JUMP_IF_FALSE   27  (to 54)                                // 23   ns
        // >>> for t in cls.traits:
        1217        10  LOAD_FAST           0   (cls)                                  // 22   ns
```

```
              12  LOAD_ATTR           2   (traits)                              // 24   ns
              14  GET_ITER                (()                                   // 43   ns

    // ============ core:758 `__iter__` ============
    // >>> return iter(self.traits)
    759         0   LOAD_GLOBAL         0   (iter)                              // 17   ns
                2   LOAD_FAST           0   (self)                              // 16   ns
                4   LOAD_ATTR           1   (traits)                            // 38   ns

        // ============= core:747 `traits` =============
        // >>> if callable(self._traits):
        750         0   LOAD_GLOBAL         0   (callable)                      // 17   ns
                    2   LOAD_FAST           0   (self)                          // 15   ns
                    4   LOAD_ATTR           1   (_traits)                       // 17   ns
                    6   CALL_FUNCTION       1   ()                              // 18   ns
                    8   POP_JUMP_IF_FALSE   12  (to 24)                         // 17   ns
        // >>> return self._traits
        752     >>  24  LOAD_FAST           0   (self)                          // 16   ns
                    26  LOAD_ATTR           1   (_traits)                       // 16   ns
                    28  RETURN_VALUE            ()                              // 38   ns
        // =============================================

                6   CALL_FUNCTION       1   ()                                  // 22   ns
                8   RETURN_VALUE            ()                                  // 42   ns
    // =============================================

        >>  16  FOR_ITER            16  (to 50)                                 // 25   ns
            18  STORE_FAST          2   (t)                                     // 22   ns
// >>> if isinstance(t, cast(type[OpTraitInvT], trait)):
1218        20  LOAD_GLOBAL         0   (isinstance)                            // 21   ns
            22  LOAD_FAST           2   (t)                                     // 21   ns
            24  LOAD_GLOBAL         3   (cast)                                  // 21   ns
            26  LOAD_GLOBAL         1   (type)                                  // 19   ns
            28  LOAD_GLOBAL         4   (OpTraitInvT)                           // 18   ns
            30  BINARY_SUBSCR           ()                                      // 22   ns
            32  LOAD_FAST           1   (trait)                                 // 19   ns
            34  CALL_FUNCTION       2   ()                                      // 41   ns

    // ============ typing:1737 `cast` =============
    // >>> return val
    1745        0   LOAD_FAST           1   (val)                               // 18   ns
                2   RETURN_VALUE            ()                                  // 43   ns
    // =============================================

            36  CALL_FUNCTION       2   ()                                      // 23   ns
            38  POP_JUMP_IF_FALSE   24  (to 48)                                 // 20   ns
1218    >>  48  JUMP_ABSOLUTE       8   (to 16)                                 // 20   ns
// >>> for t in cls.traits:
        >>  16  FOR_ITER            16  (to 50)                                 // 19   ns
            18  STORE_FAST          2   (t)                                     // 19   ns
// >>> if isinstance(t, cast(type[OpTraitInvT], trait)):
1218        20  LOAD_GLOBAL         0   (isinstance)                            // 19   ns
            22  LOAD_FAST           2   (t)                                     // 19   ns
            24  LOAD_GLOBAL         3   (cast)                                  // 19   ns
            26  LOAD_GLOBAL         1   (type)                                  // 18   ns
            28  LOAD_GLOBAL         4   (OpTraitInvT)                           // 18   ns
            30  BINARY_SUBSCR           ()                                      // 21   ns
            32  LOAD_FAST           1   (trait)                                 // 19   ns
            34  CALL_FUNCTION       2   ()                                      // 40   ns

    // ============ typing:1737 `cast` =============
    // >>> return val
    1745        0   LOAD_FAST           1   (val)                               // 18   ns
```

```
                        2   RETURN_VALUE             ()                                     // 43   ns
            // =============================================

                    36  CALL_FUNCTION        2   ()                                         // 21   ns
                    38  POP_JUMP_IF_FALSE    24  (to 48)                                    // 20   ns
        // >>> return t
        1219         40  LOAD_FAST            2   (t)                                       // 19   ns
                    42  ROT_TWO                  ()                                         // 19   ns
                    44  POP_TOP                  ()                                         // 20   ns
                    46  RETURN_VALUE             ()                                         // 47   ns
        // =============================================

                34  LOAD_CONST           3   (None)                                         // 21   ns
                36  IS_OP                1   ()                                             // 21   ns
                38  RETURN_VALUE             ()                                             // 47   ns
    // =============================================

            12  POP_JUMP_IF_TRUE     9   (to 18)                                            // 16   ns
        >>  18  LOAD_CONST           1   (None)                                             // 16   ns
            20  RETURN_VALUE             ()                                                 // 39   ns
// =============================================
```

Listing E.3: Bytecode profile trace of the original implementation of `has_trait`.

```
//// Trace of `Extensibility.time_trait_check_optimised` :

// == microbenchmarks:206 `time_trait_check_optimised` ==
// >>> has_trait = False
208          0  LOAD_CONST           1   (False)                                            // 14   ns
             2  STORE_FAST           1   (has_trait)                                        // 14   ns
// >>> for t in Extensibility.OP_WITH_REGION.traits._traits:
209          4  LOAD_GLOBAL          0   (Extensibility)                                    // 13   ns
             6  LOAD_ATTR            1   (OP_WITH_REGION)                                   // 13   ns
             8  LOAD_ATTR            2   (traits)                                           // 13   ns
            10  LOAD_ATTR            3   (_traits)                                          // 13   ns
            12  GET_ITER                 ()                                                 // 13   ns
        >>  14  FOR_ITER             12  (to 40)                                            // 13   ns
            16  STORE_FAST           2   (t)                                                // 11   ns
// >>> if isinstance(t, Extensibility.TRAIT_4):
210         18  LOAD_GLOBAL          4   (isinstance)                                       // 11   ns
            20  LOAD_FAST            2   (t)                                                // 11   ns
            22  LOAD_GLOBAL          0   (Extensibility)                                    // 11   ns
            24  LOAD_ATTR            5   (TRAIT_4)                                          // 11   ns
            26  CALL_FUNCTION        2   ()                                                 // 12   ns
            28  POP_JUMP_IF_FALSE    19  (to 38)                                            // 11   ns
210     >>  38  JUMP_ABSOLUTE        7   (to 14)                                            // 11   ns
// >>> for t in Extensibility.OP_WITH_REGION.traits._traits:
        >>  14  FOR_ITER             12  (to 40)                                            // 11   ns
            16  STORE_FAST           2   (t)                                                // 11   ns
// >>> if isinstance(t, Extensibility.TRAIT_4):
210         18  LOAD_GLOBAL          4   (isinstance)                                       // 11   ns
            20  LOAD_FAST            2   (t)                                                // 10   ns
            22  LOAD_GLOBAL          0   (Extensibility)                                    // 11   ns
            24  LOAD_ATTR            5   (TRAIT_4)                                          // 11   ns
            26  CALL_FUNCTION        2   ()                                                 // 11   ns
            28  POP_JUMP_IF_FALSE    19  (to 38)                                            // 11   ns
// >>> has_trait = True
211         30  LOAD_CONST           2   (True)                                             // 11   ns
            32  STORE_FAST           1   (has_trait)                                        // 11   ns
// >>> break
212         34  POP_TOP                  ()                                                 // 11   ns
```

```
            36  JUMP_FORWARD         1   (to 40)                                     // 11   ns
// >>> assert has_trait
213     >>  40  LOAD_FAST            1   (has_trait)                                 // 11   ns
            42  POP_JUMP_IF_TRUE     24  (to 48)                                     // 11   ns
        >>  48  LOAD_CONST           3   (None)                                      // 11   ns
            50  RETURN_VALUE             ()                                          // 23   ns
// =============================================
```

Listing E.4: Bytecode profile trace of the optimised implementation of `has_trait`.

Appendix F

Disassembly of dynamic dispatch experiments

```
main:
 stp        x29, x30, [sp, #-32]!
 str        x19, [sp, #16]
 mov        x29, sp
 ldr        x0, [x1, #8]
 mov        x19, x1
 mov        x1, xzr
 mov        w2, #0xa                          // #10
 bl        0 <strtol>
    R_AARCH64_CALL26 strtol
 ldr        x8, [x19, #16]
 mov        x19, x0
 mov        x1, xzr
 mov        w2, #0xa                          // #10
 mov        x0, x8
 bl        0 <strtol>
    R_AARCH64_CALL26 strtol
// ===== Start of function invocation ===== //
 sub        w0, w19, w0
// ====== End of function invocation ====== //
 ldr        x19, [sp, #16]
 ldp        x29, x30, [sp], #32
 ret
```

Listing F.1: Disassembly of inlined function invocation (Listing 7.1a 1).

```
main:
 stp        x29, x30, [sp, #-32]!
 str        x19, [sp, #16]
 mov        x29, sp
 ldr        x0, [x1, #8]
 mov        x19, x1
 mov        x1, xzr
 mov        w2, #0xa                          // #10
 bl        0 <strtol>
    R_AARCH64_CALL26 strtol
 ldr        x8, [x19, #16]
 mov        x19, x0
```

```
 mov        x1, xzr
 mov        w2, #0xa                          // #10
 mov        x0, x8
 bl        0 <strtol>
    R_AARCH64_CALL26 strtol
// ===== Start of function invocation ===== //
 mov        x2, x0
 add        x0, x29, #0x1f
 mov        w1, w19
 bl        0 <main>
    R_AARCH64_CALL26 Base::uninlinedFunc(int, int)
// ====== End of function invocation ====== //
 ldr        x19, [sp, #16]
 ldp        x29, x30, [sp], #32
 ret
Base::uninlinedFunc(int, int):
 sub        w0, w1, w2
 ret
```

Listing F.2: Disassembly of uninlined function invocation (Listing 7.1a 2).

```
main:
 sub        sp, sp, #0x30
 stp        x29, x30, [sp, #16]
 stp        x20, x19, [sp, #32]
 add        x29, sp, #0x10
 ldr        x0, [x1, #8]
 mov        x19, x1
 mov        x1, xzr
 mov        w2, #0xa                          // #10
 bl        0 <strtol>
    R_AARCH64_CALL26 strtol
 ldr        x8, [x19, #16]
 mov        x20, x0
 mov        x1, xzr
 mov        w2, #0xa                          // #10
 mov        x0, x8
 bl        0 <strtol>
    R_AARCH64_CALL26 strtol
 ldr        x8, [x19, #24]
 mov        x19, x0
 mov        x1, xzr
 mov        w2, #0xa                          // #10
 mov        x0, x8
 bl        0 <strtol>
    R_AARCH64_CALL26 strtol
// ===== Start of function invocation ===== //
 adrp        x8, 0 <main>
    R_AARCH64_ADR_PREL_PG_HI21 vtable for Base+0x10
 add        x11, x8, #0x0
    R_AARCH64_ADD_ABS_LO12_NC vtable for Base+0x10
 cmp        w0, #0x0
 adrp        x8, 0 <main>
    R_AARCH64_ADR_PREL_PG_HI21 vtable for Derived+0x10
 add        x8, x8, #0x0
    R_AARCH64_ADD_ABS_LO12_NC vtable for Derived+0x10
 mov        x9, sp
 add        x10, sp, #0x8
 stp        x8, x11, [sp]
 mov        w1, w20
 csel        x0, x10, x9, gt
 mov        w2, w19
```

```
 ldr        x8, [x0]
 ldr        x8, [x8]
 blr        x8
// ====== End of function invocation ====== //
 ldp        x20, x19, [sp, #32]
 ldp        x29, x30, [sp, #16]
 add        sp, sp, #0x30
 ret
Base::virtualFunc(int, int):
 sub        w0, w1, w2
 ret
Derived::virtualFunc(int, int):
 sub        w0, w2, w1
 ret
```

Listing F.3: Disassembly of polymorphic function invocation (Listing 7.1a 3).

```
//// Trace of `Extensibility.time_invoke_method_baseline` :
// == microbenchmarks:152 `time_invoke_method_baseline` ==
// >>> a = 5
154         0   LOAD_CONST          1   (5)
            2   STORE_FAST          1   (a)
// >>> b = 6
155         4   LOAD_CONST          2   (6)
            6   STORE_FAST          2   (b)
// >>> _ = None  # Simulate passing arguments
156         8   LOAD_CONST          3   (None)
            10  STORE_FAST          3   (_)
// >>> Extensibility.EXAMPLE.regularFunction
157         12  LOAD_GLOBAL         0   (Extensibility)
            14  LOAD_ATTR           1   (EXAMPLE)
            16  LOAD_ATTR           2   (regularFunction)
            18  POP_TOP                 ()
// >>> return a - b
158         20  LOAD_FAST           1   (a)
            22  LOAD_FAST           2   (b)
            24  BINARY_SUBTRACT         ()
            26  RETURN_VALUE            ()
// =============================================

// == microbenchmarks:160 `time_invoke_method` ==
// >>> a = 5
162         0   LOAD_CONST          1   (5)
            2   STORE_FAST          1   (a)
// >>> b = 6
163         4   LOAD_CONST          2   (6)
            6   STORE_FAST          2   (b)
// >>> return Extensibility.EXAMPLE.regularFunction(a, b)
164         8   LOAD_GLOBAL         0   (Extensibility)
            10  LOAD_ATTR           1   (EXAMPLE)
            12  LOAD_METHOD         2   (regularFunction)
            14  LOAD_FAST           1   (a)
            16  LOAD_FAST           2   (b)
            18  CALL_METHOD         2   ()

    // === microbenchmarks:16 `regularFunction` ====
    // >>> return a - b
     17         0   LOAD_FAST           1   (a)
                2   LOAD_FAST           2   (b)
                4   BINARY_SUBTRACT         ()
                6   RETURN_VALUE            ()
```

```
    // =============================================
            20  RETURN_VALUE            ()
// =============================================
```

Listing F.4: Bytecode trace of Python method invocation, with the CALL_METHOD and RETURN_VALUE opcodes being the target of the measurement by subtracting the elapsed time from that of the baseline inlined implementation.